

## Preface

Vectra BASIC is Hewlett-Packard's implementation of the GW<sup>TM</sup> BASIC interpreter from Microsoft Corporation. Vectra BASIC is an extension to standard BASIC and takes advantage of the facilities offered by 16-bit processors. Implemented features include full screen editing, advanced graphics, event trapping, access to non-keyboard I/O devices, and RS-232 asynchronous communications.

## Table of Contents

ii	Preface
xii	Manual Organization
xiv	Notation Conventions

### Chapter 1:

#### Getting Started

1-1	The Vectra BASIC User
1-1	Making a Working Copy of Vectra BASIC
1-2	Starting Vectra BASIC
1-2	Modes of Operation
1-2	Direct Mode
1-3	Quick Computation
1-4	Indirect Mode
1-4	Line Format
1-6	Character Set
1-8	Creating a Vectra BASIC Program
1-8	The Screen Editor
1-9	Entering a Program
1-10	Using the Alt Key
1-11	Modifying a Program
1-12	Moving the Cursor
1-13	Deleting Text and Statements
1-14	Adding Statements and Text
1-15	Edit Keys
1-18	Entering Edit Mode from a Syntax Error
1-18	Error Messages
1-18	Documenting Your Program
1-19	Printing Operations
1-19	L Commands and Statements
1-19	Using the Printer as a Device
1-20	Writing a Simple Program

## Chapter 2:

### Data, Variables, and Operators

- 2-1 Introduction
- 2-2 Constants
- 2-3 Single and Double Precision Form for Numeric Constants
- 2-4 Variables
- 2-4 Variable Names and Declaration Characters
- 2-4 Special Type Declaration Characters
- 2-4 Reserved Words
- 2-5 String Variables
- 2-5 Numeric Variables
- 2-6 Array Variables
- 2-9 Type Conversion
- 2-12 Expressions and Operators
- 2-12 Arithmetic Operators
- 2-14 Integer Division and Modulus Arithmetic
- 2-14 Overflow and Division by Zero
- 2-14 Relational Operators
- 2-15 Logical Operators
- 2-15 Functional Operators
- 2-20 String Operations
- 2-20 Concatenation
- 2-20 Comparisons

## Chapter 3:

### The Vectra BASIC Environment

- 3-1 Introduction
- 3-1 Vectra BASIC
- 3-7 Redirecting Input and Output

## Chapter 4:

### Directory and File Operations

- 4-1 Directory Paths
- 4-2 Disc File Naming Conventions
- 4-3 Disc Filenames
- 4-3 Disc Data Files
- 4-3 Sequential Files
- 4-4 Random Files
- 4-5 Creating a Random File
- 4-6 Accessing a Random File
- 4-9 Protected Files

## Chapter 5:

### Programming Tasks

- 5-1 Introduction
- 5-3 System Commands
- 5-5 Using Commands as Program Statements
- 5-6 Directory Operations
- 5-6 File Operations
- 5-8 Defining and Altering Data and Variables
- 5-9 Computer Control
- 5-10 Graphics
- 5-12 Music
- 5-13 Program Control, Branching, and Subroutines
- 5-16 Terminal Input and Output
- 5-17 RS-232 Asynchronous Communications
- 5-18 Event Trapping
- 5-19 Communications Trapping
- 5-19 Key Trapping
- 5-19 Pen Trapping
- 5-20 Play Trapping
- 5-20 Joystick Trapping
- 5-20 Timer Trapping
- 5-21 Error Trapping
- 5-22 Debugging Aids

<b>5-23</b>	Vectra BASIC Functions
<b>5-24</b>	General Purpose Functions
<b>5-24</b>	Device Sampling Functions
<b>5-25</b>	Input/Output Functions
<b>5-26</b>	Arithmetic Functions
<b>5-27</b>	Derived Functions
<b>5-29</b>	String Functions
<b>5-30</b>	Special Functions

## Chapter 6:

### GW BASIC Statements, Commands, Functions, and Variables

<b>6-1</b>	Introduction
<b>6-2</b>	Chapter Format
<b>6-3</b>	ABS Function
<b>6-3</b>	ASC Function
<b>6-4</b>	ATN Function
<b>6-5</b>	AUTO Command
<b>6-7</b>	BEEP Statement
<b>6-8</b>	BLOAD Command/Statement
<b>6-11</b>	BSAVE Command/Statement
<b>6-13</b>	CALL Statement
<b>6-16</b>	CALLS Statement
<b>6-17</b>	CDBL Function
<b>6-18</b>	CHAIN Statement
<b>6-23</b>	CHDIR Statement
<b>6-25</b>	CHR\$ Function
<b>6-26</b>	CINT Function
<b>6-27</b>	CIRCLE Statement
<b>6-31</b>	CLEAR Statement
<b>6-33</b>	CLOSE Statement
<b>6-34</b>	CLS Statement
<b>6-36</b>	COLOR Statement (text)
<b>6-39</b>	COLOR Statement (graphics)
<b>6-42</b>	COM(n) Statement
<b>6-43</b>	COMMON Statement
<b>6-45</b>	CONT Command
<b>6-47</b>	COS Function
<b>6-48</b>	CSNG Function

<b>6-49</b>	CSRLIN Function
<b>6-50</b>	CVI, CVS, CVD Functions
<b>6-51</b>	DATA Statement
<b>6-53</b>	DAT\$ Function
<b>6-54</b>	DAT\$ Statement
<b>6-56</b>	DEF FN Statement
<b>6-58</b>	DEF SEG Statement
<b>6-60</b>	DEF USR Statement
<b>6-62</b>	DEFINT/SGN/DBL/STR Statements
<b>6-64</b>	DELETE Command
<b>6-66</b>	DIM Statement
<b>6-66</b>	DRAW Statement
<b>6-73</b>	EDIT Command
<b>6-74</b>	END Statement
<b>6-75</b>	ENVIRON Statement
<b>6-77</b>	ENVIRON\$ Function
<b>6-79</b>	EOF Function
<b>6-81</b>	EOF Function (for Com Files)
<b>6-82</b>	ERASE Statement
<b>6-83</b>	ERDEV and ERDEV\$ Variables
<b>6-85</b>	ERR and ERL Variables
<b>6-88</b>	ERROR Statement
<b>6-89</b>	EXP Function
<b>6-90</b>	FIELD Statement
<b>6-93</b>	FILES Command/Statement
<b>6-95</b>	FIX Function
<b>6-96</b>	FOR...NEXT Statement
<b>6-100</b>	FRE Function
<b>6-101</b>	GET Statement
<b>6-102</b>	GET Statement (graphics)
<b>6-104</b>	GET and PUT Statements (for Com Files)
<b>6-105</b>	GOSUB...RETURN Statement
<b>6-106</b>	GOTO Statement
<b>6-109</b>	HEX\$ Function
<b>6-110</b>	IF Statement
<b>6-114</b>	INKEY\$ Function
<b>6-117</b>	INP Function
<b>6-118</b>	INPUT Statement
<b>6-122</b>	INPUT# Statement
<b>6-124</b>	INPUT\$ Function
<b>6-126</b>	INSTR Function

<b>6-127</b>	<b>INT Function</b>
<b>6-128</b>	<b>IOCTL Statement</b>
<b>6-129</b>	<b>IOCTL\$ Function</b>
<b>6-130</b>	<b>KEY Statement</b>
<b>6-134</b>	<b>KEY(n) Statement</b>
<b>6-136</b>	<b>KILL Command/Statement</b>
<b>6-138</b>	<b>LEFT\$ Function</b>
<b>6-139</b>	<b>LEN Function</b>
<b>6-140</b>	<b>LET Statement</b>
<b>6-141</b>	<b>LINE Statement</b>
<b>6-145</b>	<b>LINE INPUT Statement</b>
<b>6-146</b>	<b>LINE INPUT# Statement</b>
<b>6-148</b>	<b>LIST and LLIST Command</b>
<b>6-151</b>	<b>LOAD Command</b>
<b>6-153</b>	<b>LOC Function</b>
<b>6-154</b>	<b>LOC Function (for COM files)</b>
<b>6-155</b>	<b>LOCATE Statement</b>
<b>6-158</b>	<b>LOF Function</b>
<b>6-159</b>	<b>LOF Function (for COM files)</b>
<b>6-160</b>	<b>LOG Function</b>
<b>6-160</b>	<b>LPOS Function</b>
<b>6-161</b>	<b>LPRINT and LPRINT USING Statements</b>
<b>6-162</b>	<b>LSET and RSET Statements</b>
<b>6-164</b>	<b>MERGE Command</b>
<b>6-166</b>	<b>MID\$ Function</b>
<b>6-167</b>	<b>MID\$ Statement</b>
<b>6-168</b>	<b>MKDIR Statement</b>
<b>6-170</b>	<b>MKIS,MKS\$,MKD\$ Functions</b>
<b>6-171</b>	<b>NAME Statement</b>
<b>6-173</b>	<b>NEW Command</b>
<b>6-174</b>	<b>OCT\$ Function</b>
<b>6-175</b>	<b>ON COM Statement</b>
<b>6-177</b>	<b>ON ERROR GOTO Statement</b>
<b>6-179</b>	<b>ON...GOSUB Statement</b>
<b>6-180</b>	<b>ON...GOTO Statement</b>
<b>6-181</b>	<b>ON KEY Statement</b>
<b>6-184</b>	<b>ON PEN Statement</b>
<b>6-186</b>	<b>ON PLAY Statement</b>
<b>6-188</b>	<b>ON STRIG Statement</b>
<b>6-190</b>	<b>ON TIMER Statement</b>
<b>6-192</b>	<b>OPEN Statement</b>

<b>6-197</b>	<b>OPEN "COM Statement</b>
<b>6-201</b>	<b>OPTION BASE Statement</b>
<b>6-202</b>	<b>OUT Statement</b>
<b>6-203</b>	<b>PAIN Statement</b>
<b>6-210</b>	<b>PEEK Function</b>
<b>6-211</b>	<b>PEN Statement</b>
<b>6-212</b>	<b>PEN(n) Function</b>
<b>6-214</b>	<b>PLAY Statement</b>
<b>6-218</b>	<b>PLAY(n) Function</b>
<b>6-219</b>	<b>PMAP Function</b>
<b>6-220</b>	<b>POINT Function</b>
<b>6-222</b>	<b>POKE Statement</b>
<b>6-223</b>	<b>POS Function</b>
<b>6-224</b>	<b>PRESET Statement</b>
<b>6-226</b>	<b>PRINT Statement</b>
<b>6-229</b>	<b>PRINT USING Statement</b>
<b>6-235</b>	<b>PRINT# and PRINT# USING Statements</b>
<b>6-238</b>	<b>PSET Statement</b>
<b>6-240</b>	<b>PUT Statement</b>
<b>6-241</b>	<b>PUT Statement (graphics)</b>
<b>6-244</b>	<b>RANDOMIZE Statement</b>
<b>6-246</b>	<b>READ Statement</b>
<b>6-248</b>	<b>REM Statement</b>
<b>6-250</b>	<b>RENUM Command</b>
<b>6-252</b>	<b>RESET Command/Statement</b>
<b>6-253</b>	<b>RESTORE Statement</b>
<b>6-254</b>	<b>RESUME Statement</b>
<b>6-255</b>	<b>RETURN Statement</b>
<b>6-256</b>	<b>RIGHT\$ Function</b>
<b>6-257</b>	<b>RMDIR Statement</b>
<b>6-259</b>	<b>RND Function</b>
<b>6-260</b>	<b>RUN Command/Statement</b>
<b>6-262</b>	<b>SAVE Command</b>
<b>6-264</b>	<b>SCREEN Function</b>
<b>6-267</b>	<b>SCREEN Statement</b>
<b>6-272</b>	<b>SGN Function</b>
<b>6-273</b>	<b>SHELL Statement</b>
<b>6-276</b>	<b>SIN Function</b>
<b>6-277</b>	<b>SOUND Statement</b>
<b>6-281</b>	<b>SPACE\$ Function</b>
<b>6-282</b>	<b>SPC Function</b>

**6-283** SQR Function  
**6-284** STICK Function  
**6-285** STOP Statement  
**6-287** STR\$ Function  
**6-288** STRIG Statement  
**6-289** STRIG(n) Function  
**6-291** STRING(n) Statement  
**6-293** STRING\$ Function  
**6-294** SWAP Statement  
**6-295** SYSTEM Command/Statement  
**6-296** TAB Function  
**6-297** TAN Function  
**6-298** TIMES Function  
**6-299** TIMES Statement  
**6-300** TIMER Function  
**6-301** TIMER Statement  
**6-303** TRON/TROFF Statements  
**6-305** USR Function  
**6-306** VAL Function  
**6-307** VARPTR Function  
**6-310** VARPTR\$ Function  
**6-312** VIEW Statement  
**6-316** VIEW PRINT Statement  
**6-317** WAIT Statement  
**6-318** WHILE...WEND Statement  
**6-320** WIDTH Statement  
**6-322** WINDOW Statement  
**6-326** WRITE Statement  
**6-327** WRITE# Statement

## Appendix A:

### Error Codes and Error Messages

## Appendix B:

### Reference Tables

**B-1** Character Sets  
**B-1** Accessing the HP Character Set  
**B-2** Using CHR\$  
**B-3** Using the Alt Key  
**B-3** ASCII Character Codes  
**B-14** Scan Codes  
**B-17** Reserved Words  
**B-18** Syntax Charts

## Appendix C:

### Assembly Language Subroutines

**C-1** Introduction  
**C-2** Memory Allocation  
**C-3** CALL Statement  
**C-11** USR Function

## Appendix D:

### Installing Vectra BASIC

**D-1** Making a Working Copy of GW BASIC  
**D-2** For Dual Disc Drive Users  
**D-3** Copying Vectra BASIC  
**D-4** Making Vectra BASIC a P.A.M. Option  
**D-5** For Hard Disc Drive Users  
**D-6** Starting GW BASIC

## Appendix E:

### Differences in Versions of BASIC

## Manual Organization

Throughout this manual, the term "instruction" is a generic term that combines commands, statements, and functions under one name.

Chapter 1 introduces the Hewlett-Packard GW BASIC language and gives guidelines so you may start writing your own GW BASIC programs. It also describes the screen editor.

Chapter 2 describes general features about GW BASIC, such as data types and operations.

Chapter 3 gives specific information about the GW BASIC command line. This includes redirection of input and output.

Chapter 4 describes files and directories.

Chapter 5 groups the GW BASIC instructions according to tasks.

Chapter 6 is a comprehensive alphabetical listing of all the GW BASIC commands, statements, functions, and variables.

The appendices provide more information on error codes and error messages, your computer's terminal features, assembly-language subroutines, and installation procedures. It also supplies the necessary reference tables.

## Summary of Changes

The following list summarizes the changes between GW BASIC and Series 100/BASIC. Please see the description of each instruction for full details.

GW BASIC provides a full screen editor that lets you change any line that appears on the screen. This editing facility replaces "Modify Mode" and the "Edit Mode Subcommands" which were available in Series 100/BASIC.

GW BASIC supports several statements that accomplish tasks for which Series 100/BASIC used escape sequences. (See the discussion of Computer Control in Chapter 5.) Consequently, GW BASIC offers little or no support for escape sequences.

MS-DOS 2.0 supports pathnames. Therefore, the following commands and statements now make use of this facility:

- Commands: **BLDAD**, **BSAVE**, **KILL**, **LOAD**, **MERGE**, **NAME**, **RUN**, **SAVE**
- Statements: **CHAIN**, **OPEN**

Three new statements support the handling of directories. These statements are **CHDIR**, **MKDIR**, and **RMDIR**.

The **DELETE**, **RANDOMIZE**, and **RETURN** statements also contain new enhancements.

Enhancements have been made to the **GWBASIC** command line, including the provision to redirect standard input and output.

New statements have been added to support computer control, graphics, RS-232 asynchronous communications, and event trapping.

New functions include **POINT**, **SCREEN**, **TIMER**, **CSRLIN**, and **VARPTR**.

## Notation Conventions

The notation conventions that we use in this manual adhere to the following rules:

### CAPITAL LETTERS

You must enter those words that appear in capital letters exactly as they are shown. However, this only aids reading the syntax charts as GIV BASIC automatically shifts variable names and key words to upper case letters.

### lower case letters

Words shown in italicized, lower case letters are words that you must supply.

### [square brackets]

Square brackets enclose items that are optional.

### {braces}

Braces enclose multiple items when you must select between the available choices.

### vertical bar |

A vertical bar divides the selection of items that are enclosed by braces

### ellipsis (...)

Items that are followed by an ellipsis may be repeated any number of times (up to the length of the input line).

### punctuation

The punctuation symbols that serve special functions have been described above. You must include all other punctuation symbols (such as commas, semicolons, parentheses, quotation marks, etc.) exactly as they appear within the format charts.

Consider this example:

```
INPUT{;}[ "prompt" {; {,} variable {, variable} ]...
```

To be valid, an INPUT statement must contain the keyword INPUT and at least one variable. Since *variable* is italicized, you must replace this descriptive term with an appropriate name. Square brackets surround optional parameters. For example, the semicolon and prompt string are both optional. However, if you include a prompt, you must enclose the string in quotation marks and end the string with either a semicolon or a comma. You may list several variables, but you must separate them with commas.

## Chapter 1:

### Getting Started

1-1	The Vectra BASIC User
1-1	Making a Working Copy of Vectra BASIC
1-2	Starting Vectra BASIC
1-2	Modes of Operation
1-2	Direct Mode
1-3	Quick Computation
1-4	Indirect Mode
1-4	Line Format
1-6	Character Set
1-8	Creating a Vectra BASIC Program
1-8	The Screen Editor
1-9	Entering a Program
1-10	Using the Alt Key
1-11	Modifying a Program
1-12	Moving the Cursor
1-13	Deleting Text and Statements
1-14	Adding Statements and Text
1-15	Edit Keys
1-18	Entering Edit Mode from a Syntax Error
1-18	Error Messages
1-18	Documenting Your Program
1-19	Printing Operations
1-19	L Commands and Statements
1-19	Using the Printer as a Device
1-20	Writing a Simple Program

## Chapter 1

# 1

## Getting Started

### The Vectra BASIC User

To be a successful Vectra BASIC user, you should be familiar with general programming concepts and the BASIC language. If you are unfamiliar with BASIC, we recommend that you either read one of the introductory texts on programming in BASIC or take a beginning-level course on this language.

### Making A Working Copy Of Vectra BASIC

You should always make a working copy of your application software as a safeguard against possible damage or loss to your "master" disc. Appendix D provides details for installing Vectra BASIC on the flexible discs or a hard disc.

After you have made a working copy, you should use this copy for your daily work and store the master disc in a safe place.

## Starting Vectra BASIC

Starting Vectra BASIC depends on the choices you made when you copied your Vectra BASIC disc.

If you have installed Vectra BASIC as an option on your P.A.M. Main Menu, simply move the arrow to Vectra BASIC and press **F1 to Start Application**. Press **F1** again for **No parameters**.

If Vectra BASIC is not on the Main Menu, you have 2 choices:

1. Select **DOS COMMANDS** from the P.A.M. Menu. Insert your Vectra BASIC disk in drive A, and type **GMBASIC**.
2. Insert your Vectra BASIC disk in drive A, then press **F6**, **Show .EXE .COM .BAT**. Select Vectra BASIC from the list and press **F1 to Start Application**.

### Note

If you are using a hard disc, and have placed Vectra BASIC in a subdirectory, change to that directory (**CD C:\path**) before you press the **SHOW .EXE . . .** function key.

## Modes Of Operation

Once the Vectra BASIC interpreter assumes control, it prompts you for information by displaying the letters **OK**. This manual refers to this state (where the interpreter is awaiting your next command) as the command level. After Vectra BASIC issues its first **OK** prompt, it remains at the command level until you enter a **RUN** command.

At the command level, you may converse with the interpreter in one of two modes: Direct Mode or Indirect Mode.

### Direct Mode

Direct Mode is useful for debugging programs and for quick computations.

In Direct Mode, you do not precede Vectra BASIC statements or functions with line numbers. Rather, you "talk" interactively with the Vectra BASIC interpreter, and Vectra BASIC executes each instruction as you enter it.

For example,

```
OK
PRINT "HELLO MOM" [Enter]
HELLO MOM
OK
```

You may use Direct Mode to display the results of mathematical and logical operations (using **PRINT** statements) or to store the results for later use (using the **LET** statement). However, instructions that produce these results are lost after the interpreter executes the instruction.

**Quick Computation** You may use Vectra BASIC as a calculator to perform quick calculations without writing a program. You can perform numeric operations in Direct Mode by entering a question mark (?), then the expression. (Vectra BASIC interprets the question mark as an abbreviation for **PRINT**.) For example, to calculate two times the sum of four plus two where the sum is raised to the third power, type:

```
?2*(4+2)^3 [Enter]
```

Vectra BASIC performs the calculation and prints the result:

```
432
```

When you assign values to variables with the **LET** statement, the values are not displayed. You can only view these values by printing them to the screen. Furthermore, the values that you assigned to variables are lost when you subsequently run a program or exit Vectra BASIC.

In the following example, the two LET statements set the value for X and Y. Vectra BASIC does not display these values. The last line is a PRINT statement that displays the answer for this simple problem.

```
LET X = 3 [Enter]
OK
LET Y = -8 * X [Enter]
OK
PRINT ABS(X * Y) [Enter]
72
OK
```

## Indirect Mode

You use Indirect Mode when entering programs. In this mode, you precede each line with a unique line number, and Vectra BASIC stores these lines in your computer's memory. You then execute the program by entering the RUN command.

For example,

```
OK
10 PRINT "HELLO MOM" [Enter]
RUN [Enter]
HELLO MOM
OK
```

**Line Format** Program lines in a Vectra BASIC program have the following format:

*nnnn statement [statement]...*

*nnnn* represents a line number that may be from 1 to 5 digits in length. Permissible values range from 0 to 65529.

A program line always begins with a line number, may contain a maximum of 255 characters, and ends when you press the [Enter] key. When a line contains more than 255 characters, Vectra BASIC truncates the excess characters.

Line numbers indicate the order in which Vectra BASIC stores the line in memory. They must be whole numbers. Numbers also serve as labels for branching and editing.

You may use a period with the EDIT, LIST, AUTO, and DELETE commands to refer to the current line. For example, EDIT . displays the last referenced or entered line for editing. Notice that you must include a space between the name of the command and the period.

A program line may contain a maximum of 255 characters. You may accomplish this in one of two ways. The simpler procedure is to type continuously, without pressing the [Enter] key. However, if you want to "format" the line (for example, put the THEN and ELSE parts of an IF statement on separate lines), you may end a screen line by pressing [CTRL] [J]. This generates a line feed character which moves you to the next screen line without terminating the logical line. A logical line is a string of text that Vectra BASIC treats as a unit. When you finish typing the logical line, pressing the [Enter] key ends the line at that point.

## Note

You must always end the last screen line of a logical ("program") line by pressing the [Enter] key.

*statement* is any legal Vectra BASIC instruction.

A statement is either executable or non-executable. Executable statements instruct Vectra BASIC to perform a specific action. For example, LET P1 = 3.141593 is an executable statement. They DATA and REM statements are non-executable statements. They result in no visible action by Vectra BASIC when Vectra BASIC encounters them.

You may enter multiple statements on one line, but you must separate each statement with a colon (:).

## Character Set

The Vectra BASIC character set contains the alphabetic characters, numeric characters, and a selected set of special symbols.

Alphabetic characters are either upper-case or lower-case letters.

Numeric characters are the decimal digits 0 through 9.

Table I-1 lists the special characters that Vectra BASIC supports.

Table I-1. Vectra BASIC Special Characters

Character	Description
	Blank
=	Equal sign or assignment symbol
+	Plus sign or concatenation symbol
-	Minus sign
*	Multiplication sign or asterisk
/	Division sign or slash character
\	Integer division symbol or backslash
^	Exponentiation symbol or caret
%	Percent sign or integer type declaration character
!	Exclamation point or single-precision type declaration character
#	Number sign or double-precision type declaration character
\$	Dollar sign or string type declaration character
(	Left parenthesis
)	Right parenthesis
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
:	Semicolon
:	Colon or program statement separator
&	Ampersand
?	Question mark
<	Less than symbol
>	Greater than symbol
@	At sign
_	Underscore
'	Apostrophe or remark delimiter
"	Quotation mark or string delimiter

## Creating A Vectra BASIC Program

The first step in creating a Vectra BASIC program is entering the necessary text into computer memory. Vectra BASIC provides a convenient environment for this task through its screen editor. As the editor provides a variety of editing features, you should experiment with the various options until you find the methods that you like best. (For an example of some of these features, see the discussion under "Writing A Sample Program".)

The section on "Edit Keys" provides more information on program editing.

### The Screen Editor

The Vectra BASIC program editor is a screen line editor. That is, you can change a line anywhere on the screen, but you can only change one line at a time. You record the changes to a line by pressing the **Enter** key while the cursor is anywhere within that line.

### Note

You need not move the cursor to the end of a logical line before you press the **Enter** key. The editor "knows" where each line ends, and it processes the entire line, regardless of the cursor's position when you press the **Enter** key.

The screen editor processes all text that you type while Vectra BASIC is at the command level.

Vectra BASIC considers any line of text that begins with a number to be a program statement. It then takes one of the following actions:

- adds a new line to the program if the line doesn't currently exist
- replaces the line if the line does exist
- deletes an existing line if you enter only the line number
- displays an error message if:
  - you attempt to delete a nonexistent line
  - program memory is exhausted
  - a major syntax error is discovered

You enter a program by simply typing the required text. As you type the characters, the editor interprets each keystroke.

You may use this feature to reduce your typing. For example, the editor interprets a question mark (?) as the reserved word **PRINT**.

### Entering A Program

Using the Alt key

Some of the most frequently used Vectra BASIC instructions can be entered by pressing the **[Alt]** key, and the first letter of the instruction. Hold down the **[Alt]** key, and then press the alphabetic key. A few letters have no associated Vectra BASIC instruction.

Alt +	Result:
A	AUTO
B	BSAVE
C	COLOR
D	DELETE
E	ELSE
F	FOR
G	GOTO
H	HEX\$
I	INPUT
J	(none)
K	KEY
L	LOCATE
M	MERGE
N	NEXT
O	OPEN
P	PRINT
Q	(none)
R	RUN
S	SCREEN
T	THEN
U	USING
V	VAL
W	WIDTH
X	XOR
Y	(none)
Z	(none)

The **[Alt]** key can also be used if you want to enter some of the special characters that aren't found on the keys. To print these letters, hold the **[Alt]** key, type the one, two or three numbers from the ASCII chart in Appendix B, and then release the **[Alt]** key. You must release the **[Alt]** key between each special character you type.

Example:

The Vectra BASIC instruction to print the word "Höhe" can be entered by typing this sequence of keys:

```
Alt P
"
H
Alt 148
You must hold Alt, type the 3 numbers on the
numeric keypad, then release [Alt] .
h
e
"
[Enter]
```

The function **CHR\$** can also be used to print these special characters. The equivalent Vectra BASIC instruction would be:  
**PRINT "H";CHR\$(148);"he"**

Modifying A Program

You may choose between two methods to modify a line that resides in your computer's memory. If the line currently appears on the screen, you can use the editor to modify the line directly. When the line is not displayed, you can use either the **EDIT** command to list a specific line or the **LIST** command to list a portion of the program. You may then modify the displayed lines by using the screen editor.

Note

You must remember that your computer's screen is only a viewing window into computer memory. While you may alter the text that appears on the screen, the text that is stored in computer memory remains unchanged until you press the **[Enter]** key. If you are modifying several lines, you must press the **[Enter]** key on each line where a change occurs.

**Moving The Cursor** Before you can make changes to existing text, you must move the cursor to the proper position.

### Note

The editor provides four keys that move the cursor one-character position at a time. These keycaps have solid triangles that indicate the direction in which the cursor moves, but this discussion uses descriptive "titles" for each keycap.

You may move the cursor through a line of text by pressing the Cursor-left or Cursor-right key. When the cursor reaches the end of the current line, it will automatically wrap around to the next line.

### Caution

You should avoid using the **Space bar** to move the cursor. If you try positioning the cursor in this manner, the editor replaces the existing text with blanks.

You may move the cursor vertically by pressing the Cursor-up or the Cursor-down key. These keys move the cursor through all the lines listed on the screen.

The editor also provides keys that move the cursor over larger blocks of text.

Pressing the **Tab** key moves the cursor to the next tab stop. (Vectra BASIC automatically sets tab stops at every eighth character position.)

Pressing **CTRL** plus Cursor-Left or Cursor-Right moves the cursor between "words." A word is a character or group of characters that begins with a letter or number. Spaces or special characters separate words.

Simultaneously pressing **CTRL** and Cursor-Right moves the cursor to the beginning of the next word. Simultaneously pressing **CTRL** Cursor-Left moves the cursor to the previous word.

In some instances, you may want to add text to the end of a line. Pressing **End** moves the cursor to the end of the logical line. (Remember, a logical line may consist of several "screen" lines, up to a maximum of 255 characters.)

**Deleting Text And Statements** The editor provides several keys for deleting existing text.

Pressing the **DEL** key erases the character directly above the cursor.

You may truncate a line (that is, erase all text from the cursor's current position to the end of the line) by simultaneously pressing the **CTRL** **End** keys. You can use this editing feature when you want to keep the same line number in a program, but change all the text on the line.

You may erase the entire line (regardless of the cursor's position) by pressing **ESC**. This erases the lines from the screen, but does not erase the line from the computer's memory. (You may use the **DELETE** command to remove any unwanted lines from a program.)

You may erase the entire screen by simultaneously pressing **[CTRL]** **[Home]**. (Remember erasing text from the screen has no effect on the information stored in computer memory.)

If your intent, however, is to clear computer memory, you should use the **NEW** command. The **NEW** command deletes the program that currently resides in computer memory. You normally use this command before you begin entering a new program.

**Adding Statements And Text** Besides deleting text, you may also add text to a program.

You add lines of text by assigning a new line number and typing in the required information.

You may add characters to an existing line by putting your cursor correctly in Insert Character mode. First, you must position the cursor correctly in the line that you wish to modify. Then you press the **[ins]** key. The cursor will change shape. From this point, any characters you type are inserted before the character marked by the modified cursor. The existing characters on the line shift to the right to make room for the added characters. When these shifted characters reach the edge of the screen, inserting more text creates a new line below the line which you are editing, and the "excess" characters wrap onto this line.

If you press the **[Tab]** key while in Insert Character mode, the editor inserts the number of spaces needed to move the cursor to the next tab stop.

You end Insert Character mode by pressing the **[insert char]** key a second time. That is, this key is a toggle switch that turns Insert Character mode on and off.

Other editing keys also end Insert Character mode. These include **[Enter]**, **[DEL]**, and the arrow keys.

## Edit Keys

This section summarizes the edit keys that Vectra BASIC supports. Where the list shows multiple keys, you must press all the keys simultaneously to implement that function.

Keys	Function
<b>[Home]</b>	Moves the cursor to the upper left corner (row 1, column 1) of the screen.
<b>[CTRL]</b> <b>[Home]</b>	Moves the cursor to the upper left corner of the screen, and erases the screen.
<b>[Backspace]</b>	Moves the cursor back one space and deletes that character.
<b>[DEL]</b>	Erases the character above the cursor. When a logical line extends off the screen's physical boundaries, the character at the left margin of the subsequent lines moves up to the previous line.
<b>[↑]</b>	Moves the cursor up one row.
<b>[↓]</b>	Moves the cursor down one row.
<b>[←]</b>	Moves the cursor left one column.
<b>[→]</b>	Moves the cursor right one column.
<b>[CTRL]</b> <b>[←]</b>	Moves the cursor left one word.
<b>[CTRL]</b> <b>[→]</b>	Moves the cursor right one word.
<b>[End]</b>	Moves the cursor to the end of the logical line.

## Keys

**CTRL** **End**

Erases all characters from the cursor position to the end of the logical line.

**ESC**

Erases a logical line from the screen (but not from the computer's memory).

**Ins**

Toggles Insert Character mode. In Insert Character mode, characters following the cursor move right as the editor inserts new characters at the cursor's current position.

**Tab**

Moves the cursor to the next tab stop without affecting characters on the screen. When the editor is in Insert Character mode, pressing the Tab key inserts the necessary number of spaces (blank characters) to move the cursor from the cursor's current position to the next tab stop.

## Function

The following list summarizes the control characters that Vectra BASIC supports.

**CTRL** **B**

Moves the cursor to the beginning of the previous word.

**CTRL** **E**

Erases text to the end of line.

**CTRL** **F**

Moves the cursor to the beginning of the next word.

**CTRL** **G**

Rings the computer's bell.

**CTRL** **H**

Backspaces over (and deletes) the previous character.

**CTRL** **I**

Moves the cursor to the next tab stop. Tab stops are set at every eighth character position.

**CTRL** **J**

Line feed with carriage return.

**CTRL** **K**

Homes the cursor to the upper left corner of the screen.

**CTRL** **L**

Erases the entire screen.

**CTRL** **M**

Carriage return

**CTRL** **N**

Moves the cursor to end of the logical line.

**CTRL** **R**

Toggles Insert Character mode.

**CTRL** **T**

Displays the next set of function keys in 40 character mode, or toggles function key display.

**CTRL** **A**

Moves the cursor up one row.

**CTRL** **D**

Moves the cursor down one row.

**CTRL** **I**

Moves the cursor left one column.

**CTRL** **O**

Moves the cursor right one column.

**CTRL** **Break**

Stops program execution and returns control to Vectra BASIC command level.

**CTRL**

Pauses program execution (or LISTING of a program.) Any key on the keyboard resumes the activity.

**Num Lock**

## Entering Edit Mode From A Syntax Error

When Vectra BASIC encounters a syntax error while executing a program, it automatically enters Edit mode at the line that caused the error. For example,

```
10 K=2(4) [Enter]
RUN [Enter]
Syntax error in 10
OK
10 K=2(4)
```

The cursor is positioned within the displayed line at the point where Vectra BASIC could not interpret the instruction. In the example above, the cursor is positioned on the open parenthesis.

When you finish editing a line, pressing the **[Enter]** key directs Vectra BASIC to incorporate all the changes into that line. However, modifying a line causes all variable values to be lost. If you want to preserve variable values for further examination, you should press **[CTRL] [Break]** before attempting to modify any lines. This action returns Vectra BASIC to the command level and preserves all variable values.

## Error Messages

When the Vectra BASIC interpreter detects a fatal error (that is, one that halts program execution), it prints an appropriate error message. Appendix A provides a complete list of error codes and their meanings.

## Documenting Your Program

As a general rule for writing good programs in Vectra BASIC, we recommend that you include plenty of comment lines to document the program properly. See the **REM** statement for further information.

## Printing Operations

You may choose between these three methods for accessing a printer from Vectra BASIC:

If your Vectra is equipped with a parallel printer, you may use Shift-Print Screen to print the screen contents to the printer.

You can use the Vectra BASIC "L" commands and statements.

You can use the **OPEN** statement to open the printer as a device.

## L Commands And Statements

The L commands and statements print to the MS-DOS primary list device (the default is LPT1). The L commands are:

<b>LLIST</b>	Prints a program listing directly to the printer.
<b>LPRINT</b>	Prints information that is supplied by a program.
<b>LPRINT USING</b>	Formats and prints information that is supplied by a program.

## Using the Printer as a Device

This option provides flexibility within a program. You can choose whether to send output to the screen, a printer, or a file. This example demonstrates one simple method:

```
100 INPUT "Printer or Screen"; D$
110 DV$="--SCRN:"; IF D$="P" THEN DV$="LPT1:"
120 OPEN DV$ FOR OUTPUT AS #1
130 PRINT #1, "THIS OUTPUT IS SENT TO "; DV$
```

You can use **PRINT #**, **PRINT # USING**, and **WRITE #** when you have opened the printer or screen as a device.

## Writing A Simple Program

You need a working knowledge of several commands to start programming in Vectra BASIC. The following discussion describes these commands in their simplest form. They represent the rudimentary commands that you need to begin working with the Vectra BASIC interpreter.

<b>AUTO</b>	Generates line numbers automatically when you press the <b>[Enter]</b> key. You may end this feature by simultaneously pressing the <b>[CTRL]</b> and <b>[Break]</b> keys.
<b>LIST</b>	Displays all or part of a program on the computer's screen.
<b>DELETE</b>	Removes one or more lines from a program.
<b>RENUM</b>	Changes the numbering of the lines in a program.
<b>RUN</b>	Executes a program.
<b>SAVE</b>	Stores a copy of a program in a file on disc.
<b>FILES</b>	Lists the names of all the files on the current disc directory.
<b>KILL</b>	Deletes a file from the disc.
<b>NEW</b>	Clears the program that is currently stored in your computer's memory. This frees memory so you may use the area for other purposes, such as starting a new program.
<b>SYSTEM</b>	Leaves Vectra BASIC and returns control to the operating system.
<b>[CTRL] [Break]</b>	Stops execution and returns control to the Vectra BASIC command level.

The following steps lead you through a simple exercise where you use each of these commands.

1. Turn on your system and insert the work disc that contains your copy of Vectra BASIC.
2. When the P.A.M. screen appears, select Vectra BASIC, then press **[Start Applic]**.
3. After P.A.M. loads Vectra BASIC into memory, a message, similar to the following, appears:

```
Vectra BASIC Version 3.12
Copyright (c) Microsoft 1983, 1984, 1985
Copyright (c) Hewlett Packard 1984, 1985
Compatibility Software
Copyright (c) Phoenix Software Associates
Ltd. 1984, 1985
xxxxx Bytes free
0k
```

where:

**xxxxx** is the number of bytes available in memory for programs and data.

**0k** is the Vectra BASIC prompt. Whenever this prompt appears, Vectra BASIC is waiting for your next command.

4. To start programming, type:

```
AUTO [Enter]
```

Hereafter, Vectra BASIC automatically prompts you with line numbers. The first number to appear is 10.

5. Now type the following program:

```
10 FOR I = 1 TO 10 [Enter]
20 PRINT I [Enter]
30 NEXT I [Enter]
40 PRINT "LOQP DONE, I = "; I [Enter]
50 END [Enter]
60
```

6. Simultaneously press the **CTRL** and **Break** keys to stop the line number prompt.

7. To see what output this program produces, type the command:

```
RUN [Enter]
```

The program prints the following display to your screen:

```
1
2
3
4
5
6
7
8
9
10
LOOP DONE, I = 11
Ok
```

8. To see a complete listing of the program on your screen, type:

```
LIST [Enter]
```

Vetra BASIC responds by printing:

```
10 FOR I = 1 TO 10
20 PRINT I
30 NEXT I
40 PRINT "LOOP DONE, I = "; I
50 END
Ok
```

9. The screen editor provides a variety of ways to modify an existing program. This step shows one way to change the first line of the program so the loop proceeds backwards from 10 to 1.

- Move the cursor to line 10 by repeatedly pressing the Cursor-up key.
- Move the cursor to the number 1 (after the equal sign) by repeatedly pressing the Cursor-right key.

- Press **CTRL** **END** to erase the remainder of the line.
- Complete the FOR statement by typing:

```
10 TO 1 STEP -1 [Enter]
```

10. Press **CTRL** **Home** to clear the screen and position the cursor to row 1, column 1.

11. List the program by using the LIST command. Notice that you may also press function key 1 and the **Enter** key to list the program.

Vetra BASIC responds by printing:

```
10 FOR I = 10 TO 1 STEP -1
20 PRINT I
30 NEXT I
40 PRINT "LOOP DONE, I = "; I
50 END
Ok
```

12. Use the RUN command to see how your changes have affected program execution.

The following display appears on your screen:

```
10
9
8
7
6
5
4
3
2
1
LOOP DONE, I = 0
Ok
```

- 13.** This step shows how you can use the screen editor to delete and reenter lines of text:
- To set-up your work area, press **CTRL** **[Home]** to erase the entire screen, then use the **LIST** command to list your program.
  - Delete line 40 by typing:  
40 **[Return]**
  - **LIST** the program to see the results.
- If you change your mind and decide you want to keep line 40, you needn't retype the line as long as a copy of the line appears on your screen.
- Use the Cursor-up key to move the cursor to the original version of line 40 in the first listing of the program.
  - Once the cursor is within this line, pressing the **[Enter]** key restores the entire line.
  - **LIST** the program to verify that the program is back to its original state.
- 14.** You may also delete line 40 by typing:  
DELETE 40 **[Enter]**
- LIST** your program again and notice that Vectra BASIC has deleted line 40 from the program.
- 15.** If you wish to have the program lines in sequential order, renumber the lines by typing:  
RENUM **[Enter]**
- Listing the program shows that the line numbers have been renumbered starting with 10 and incrementing by 10 at each step.

- 16.** You may use the **SAVE** command to write your programs to a disc file. For example, if you want to name the program **PRG1**, type:

```
SAVE "n:PRG1" [Enter]
```

Since the name for the program is a character string, you must surround the name with quotation marks.

Additionally, since you did not specify a disc drive or directory name, Vectra BASIC stores the file on the currently selected, default drive, in the current directory.

To save the program on a different disc, type:

```
SAVE "n:PRG1" [Enter]
```

Here, **n**, names the disc drive that you selected. If you selected drive **C**, for example, the command appears as:

```
SAVE "C:PRG1"
```

To specify a different drive and directory, type:

```
SAVE "n:\PROGRAMS\PRG1"
```

### Note

This command will generate an error message if the directory "PROGRAMS" does not exist on the specified drive.

Vectra BASIC supplies the MS-DOS file type, **.BAS** for you. After it has successfully written your file to disc, Vectra BASIC responds with its **OK** prompt.

- 17.** To see a listing of all the files on the default disc directory (including the one you just saved), type:

```
FILES [Enter]
```

- 18.** If you want to delete your program file from the default disc directory, type:

```
KILL "PRG1.BAS" [Enter]
```

### Note

When using the **KILL** command, you must supply the file type **.BAS**. Vectra BASIC provides no default file extension for you.

19. If you want to erase the program file from your computer's memory, type:  
**NEW** **[Enter]**  
 This clears the memory area for Vectra BASIC so you can enter a new program.

**Note**

Using the **NEW** command does not clear the file from your disc.

20. When you are ready to leave Vectra BASIC and return control to the operating system, type:

**SYSTEM** **[Enter]**

**Note**

Before exiting, be sure to **SAVE** your program if you wish to use it again.

**Chapter 2:**

**Data, Variables, and Operators**

2-1	Introduction
2-2	Constants
2-3	Single and Double Precision Form for Numeric Constants
2-4	Variables
2-4	Variable Names and Declaration Characters
2-4	Special Type Declaration Characters
2-4	Reserved Words
2-5	String Variables
2-5	Numeric Variables
2-6	Array Variables
2-9	Type Conversion
2-12	Expressions and Operators
2-12	Arithmetic Operators
2-14	Integer Division and Modulus Arithmetic
2-14	Overflow and Division by Zero
2-15	Relational Operators
2-15	Logical Operators
2-19	Functional Operators
2-20	String Operations
2-20	Concatenation
2-20	Comparisons

# 2

## Data, Variables, and Operators

### Introduction

This chapter discusses both data representation and also the mathematical and logical operators that Vectra BASIC provides.

Numeric values may be integers, single-precision numbers, or double-precision numbers. Vectra BASIC stores all numeric values in binary representation:

- Integers require two bytes of memory storage
- Single-precision numbers require four bytes of memory storage
- Double-precision numbers require eight bytes of memory storage

An integer value may be any whole number between  $-32768$  and  $+32767$ .

Vectra BASIC stores single-precision numbers with 7 digits of precision (or 24 bits of precision), and prints up to seven digits, although only six digits may be accurate.

Vectra BASIC stores double-precision numbers with 17 digits of precision (or 56 bits of precision), and prints the number with up to 16 decimal digits.

## Constants

The actual values that Vectra BASIC uses during program execution are called constants. Constants may be numeric values or string values.

A string constant is a sequence of up to 255 alphanumeric characters that are enclosed between quotation marks. Examples of string constants are:

```
"HELLO"  
"Linda Kay"  
"$75,000.00"
```

Numeric constants are positive or negative numbers. In Vectra BASIC, numeric constants never contain commas.

There are five types of numeric constants:

- |                       |                                                                                                                                                                                                                                                                                                                                                                                                            |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Integer constants     | Integer constants are whole numbers between -32768 and +32767. They never contain decimal points.                                                                                                                                                                                                                                                                                                          |
| Fixed point constants | Fixed point constants are positive or negative real numbers (that is, numbers that contain decimal points). For example, 1.0 is a fixed point constant; not an integer constant.                                                                                                                                                                                                                           |
| Floating point        | Floating point constants are positive or negative numbers that are given in exponential form (similar to scientific notation.) A floating point constant consists of an optionally signed integer or fixed point number (the mantissa), followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10E-38 to 1.701412E+38. For example, |

```
235.988E-7 = .0000235988  
23598E = 2359000000
```

(Double-precision floating point constants use the letter D instead of E, as in 235.988D7.)

### Hex constants

Hexadecimal numbers use a Base-16 numeric system. The letters A through F correspond to the numbers 10 through 15. You must prefix hexadecimal numbers with the symbols #H. For example,

```
#HFF  
#H32F
```

### Octal constants

Octal numbers use a Base-8 numeric system. To signify an octal number, you must precede the number with an #O or #. For example,

```
#O347  
#777
```

## Single and Double Precision Form for Numeric Constants

A single-precision constant is any numeric constant that has:

- seven or fewer digits: 46.8
- exponential form using E: -1.09E-06
- a trailing exclamation point (!): 3.141593!

A double-precision constant is any numeric constant that has:

- eight or more digits: 345692811
- exponential form using D: -1.09432D-06
- a trailing number sign (#): 3.141593#

## Variables

Variables are names that represent values within a Vectra BASIC program. You may explicitly assign the value to a variable (for example, by using the LET statement). A variable may also obtain a value as the result of a computation (for example, `AREA = PI * RADIUS^2`). Vectra BASIC assumes all numeric variables have the value of zero and all string variables have the value of the null string until you actually assign them a value.

### Variable Names and Declaration Characters

Variable names in Vectra BASIC may contain a maximum of 40 characters. Allowable characters are letters, the decimal digits, and a period. The first character must be a letter. The last character may be a type declaration character (either `!`, `%`, or `$`).

Examples of valid variable names are:

```
PAGELENGTH
SALES.1983
OUTER.LIMIT
```

Vectra BASIC would reject the following variable names:

```
A.HORRENDOUSLY.LONG.VARIABLE.NAME.FOR.THE.
VALUE.OF.PAGELENGTH exceeds the limit of 40 characters.
1983SALES starts with a digit. The first character must be a
letter.
```

`OUTER.LIMIT` contains an embedded space.

### Special Type Declaration Characters

Vectra BASIC recognizes several special type declaration characters and reserved words.

**Reserved Words** Reserved words include all Vectra BASIC commands, statements, function names, and operator names. Appendix B provides a complete list of Vectra BASIC reserved words.

A variable name may not be a reserved word, but can contain embedded reserved words. For example, `LOG` and `WIDTH` are both Vectra BASIC reserved words, but `LOG.WIDTH` is a valid variable name.

Vectra BASIC assumes that a series of characters beginning with the letters `FN` is a call to a user-defined function. Therefore, you should never use these characters as the first two letters of a variable's name.

**String Variables** You may designate string variable names with a dollar sign (`$`) as the last character, or you may declare them in a `DEFSTR` statement.

For example,

```
TITLE$
```

or

```
10 DEFSTR T
20 TITLE = "1983 Sales Report"
```

The dollar sign is a variable type declaration character. It "declares" that the variable represents a string. See Chapter 6 for a full discussion of the `DEFSTR` statement.

**Numeric Variables** Numeric variable names may declare themselves to be integer, single-precision, or double-precision values. The type declaration characters for these variables names are:

```
! Integer variable
% Single-precision variable
$ Double-precision variable
```

The default type for a numeric variable name is single precision.

Examples:

- PI#** Declares **PI** to be a double-precision variable
- MAX#** Declares **MAX** to be a single-precision variable
- COUNT%** Declares **COUNT** to be an integer variable
- LENGTH** Defaults to a single-precision variable

Vectra BASIC provides another method for declaring numeric variable types. This involves using the Vectra BASIC statements **DEFINT** to define integer variables, **DEFSGN** to define single-precision variables, and **DEFDBL** to define double-precision variables.

### Array Variables

An array is a group of values (or a table) that you reference with a single variable name. The individual values in the array are called elements. You refer to each element by using the array's name and a subscript. The subscript may be an integer or an integer expression.

You declare an array by dimensioning it. You normally do this with the **DIM** statement. For example, **DIM ID\$(11)** creates a one-dimensional, string array called **ID\$**. Eleven is the index number for the "last" element of the array. When no **OPTION BASE** statement has executed, the "first" element of the array is **ID\$(0)**. Therefore, this **DIM** statement creates an array of twelve elements. Each element is a variable-length string. An implicit act of declaring an array is assigning initial values for each array element. Vectra BASIC sets the elements of a string array equal to the null string (that is, the "empty" string or a string with zero length).



As another example, consider the statements:

```
OPTION BASE 1
DIM SALES(3,4)
```

These statements also create an array of twelve elements, but in this case the elements are grouped together in 3 rows of four columns each. (The columns could represent the four fiscal quarters of a year, and the rows could represent the years 1981 to 1983.) Since the array name has no type declaration character, Vectra BASIC sets the elements of the array to single-precision numbers and assigns the value of zero to each element.

SALES (1,1)	SALES (1,2)	SALES (1,3)	SALES (1,4)
SALES (2,1)	SALES (2,2)	SALES (2,3)	SALES (2,4)
SALES (3,1)	SALES (3,2)	SALES (3,3)	SALES (3,4)

An array variable name has as many subscripts as there are dimensions in the array. For example, when **OPTION BASE 1** is used, **VECTOR(10)** refers to the tenth value in a one-dimensional array, and **MATRIX(1,4)** refers to the fourth element in the first row of a two-dimensional array.

The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

With the exception of the type declaration character `T`, you must always include the character as part of a variable name. Vectra BASIC considers variables with different type declaration characters to be different variables. This example demonstrates that `A` and `A!` are considered to be the same floating point variable:

```
A=23:A!=37:A!=45:A!=64:A(2)=87:A$="TESTING"
OK
PRINT A; A!; A$; A$; A(2); A$
37 37 45 64 87 TESTING
OK
```

A type declaration character always supercedes a `DEF` type definition:

```
DEFINT R
OK
RA0T0!=94.8:R0A0 = 101.234
OK
PRINT RA0T0!, R0A0
94.8 101
OK
```

## Type Conversion

When necessary, Vectra BASIC can convert a numeric constant from one type to another. The following examples illustrate the rules and operation of this automatic conversion.

1. When a numeric variable of one type is set equal to a numeric constant of a different type, Vectra BASIC stores the number as it was declared in the variable name. For example:

```
10 ROUND% = 23.42
20 PRINT ROUND%
30 ROUND% = 23.55
40 PRINT ROUND%
RUN
23
24
```

Setting a string variable equal to a numeric value, or vice versa, results in a `Type mismatch` error.

2. When evaluating an expression, Vectra BASIC converts all operands in an arithmetic or relational operation to the degree of precision of the most-precise operand. Vectra BASIC also calculates the result to this degree of precision.

Consider these examples:

- Vectra BASIC performs the following calculation in double-precision arithmetic because the numerator is given as a double-precision number. Vectra BASIC also stores the result as a double-precision value.

```
10 TWO_THIRDS# = 2# / 3
20 PRINT TWO_THIRDS#
RUN
.6666666666666667
```

- b.** Vectra BASIC performs the following calculation in double-precision arithmetic because the numerator is given as a double-precision number. Since the variable is a single-precision variable (by default), Vectra BASIC rounds the result and stores the value as a single-precision value:

```
10 TWO_THIRDS = 2# / 3
20 PRINT TWO_THIRDS
RUN
.6666667
```

- c.** Logical operators convert their operands to integers and return an integer result. Operands must be in the range of  $-32768$  to  $+32767$ , or an overflow error occurs.

```
10 FALSE = 0
20 PRINT FALSE
30 PRINT NOT FALSE
40 TRUE = 99.44
50 PRINT NOT TRUE
60 PRINT TRUE AND FALSE
RUN
0
-1
-100
0
0k
```

- d.** When a floating point value is converted to an integer, Vectra BASIC rounds the fractional portion.

```
10 COMPROMISE# = 5.2348E3
20 PRINT COMPROMISE#
RUN
5235
.
10 COMPROMISE# = 5.2342E3
20 PRINT COMPROMISE#
RUN
5234
```

- e.** When you assign a single-precision value to a double-precision variable, only the first seven digits, rounded, of the converted number are valid. This happens because only seven digits of accuracy were supplied with the single-precision value. The absolute value of the difference between the printed double-precision number and the original single-precision value is less than  $6.3E-8$  times the original single-precision value. For example,

```
10 PI = 3.141593
20 BAPPI# = PI
30 PRINT PI, BAPPI#
RUN
3.141593 3.141592979431152
```

- f.** When either assigning values to variables or printing results, you must be aware of the inherent problem that the computer encounters as it tries to represent decimal digits in a binary format.

Consider the equation  $2.6 * 12$  which yields a value of  $31.2$ . Then, if you subtract  $0.2$ , the mathematical result is  $31$ .

For example, the statement

```
PRINT 2.6 * 12 - 0.2
```

prints the value  $31$ .

However, the statement

```
PRINT INT(2.6 * 12 - 0.2)
```

prints the value  $30$ .

This happens since the internal representation of the result (in double precision) is  $30.99999809265137$ . Because the INT function returns the largest integer less than the numeric expression, the printed value is  $30$ .

## Expressions and Operators

An expression may be a string or numeric constant, or a variable; or it may be a combination of constants and variables with suitable operators to produce a single value.

Operators perform mathematical or logical operations on values. Vectra BASIC provides the following four categories of operators:

- Arithmetic
- Relational
- Logical
- Functional

### Arithmetic Operators

Table 2-1 lists the common arithmetic operators. See the next section for details on integer division and modulus arithmetic.

**Table 2-1. Vectra BASIC Arithmetic Operators**

Operator	Operation	Sample Expression
^	Exponentiation	RADIUS^2
-	Negation	-DEBITS
*	Multiplication	BASE * HEIGHT
/	Point Division	AREA / PI
+	Addition	WAGES + DIVIDENDS
-	Subtraction	INCOME - TAXES

Vectra BASIC evaluates an expression based upon the order of precedence of the included operators. Exponentiation is evaluated first, followed by negation. Next, any multiplication or division is performed, and finally, all addition or subtraction operations are performed. In the case of multiple operators with equal precedence, Vectra BASIC evaluates the expression from left to right.

You may change the order of evaluation by using parentheses. Vectra BASIC first evaluates all operations within parentheses. (Within a parentheses grouping, the order of precedence shown above is maintained.) Consider these examples:

Without parentheses:  $4^3 * 2 = 4096$

With parentheses:  $4^3(3^2) = 262144$

The following expanded version of the first example uses parentheses to show the implicit grouping of operations by supplying all parentheses.

$((4^3)^2) = (64)^2 = 4096$

The following list shows how you would write algebraic expressions in Vectra BASIC.

Algebraic Expression	Vectra BASIC Expression
$X + 2Y$	$X + 2 * Y$
$X - \frac{Y}{Z}$	$X - Y / Z$
$\frac{XY}{Z}$	$X * Y / Z$
$\frac{X + Y}{Z}$	$(X + Y) / Z$
$X^2 Y$	$X^2 * Y$
$X^2$	$X^2$
$X(-Y)$	$X * (-Y)$

### Note

You must always separate two consecutive operators by parentheses.

**Integer Division and Modulus Arithmetic** You specify the integer division operation with a backslash (\). With integer division, Vectra BASIC rounds the operands to integers before it performs the division. It then truncates the quotient to an integer value. (The operands must be within the range -32768 to +32767.) For example,

10 \ 4 = 2  
25.68 \ 6.99 = 3

In the order of precedence, integer division follows multiplication and floating point division.

You specify modulus arithmetic with the MOD operator. The MOD operator returns the remainder from an integer division operation. For example,

10 MOD 4 = 2 (10 \ 4 = 2 with a remainder of 2)  
25.68 MOD 6.99 = 5 (26 \ 7 = 3 with a remainder of 5)

The precedence of modulus arithmetic is just after integer division.

**Overflow and Division by Zero** When Vectra BASIC is evaluating an expression, if it encounters a zero divisor, it displays a `Division by zero` error message, sets the result to machine infinity with the sign of the numerator, and continues program execution. If the evaluation of an exponentiation results in zero being raised to a negative power, Vectra BASIC again displays the `Division by zero` error message, sets the result to positive machine infinity, and continues program execution.

If Vectra BASIC encounters a number whose absolute value is too large for it to store, it displays the `Overflow` error message, sets the result to machine infinity with the appropriate sign, and continues program execution.

Machine infinity is approximately equal to  $1.7 \times 10^{38}$ .

## Relational Operators

Relational operators compare values or variables. The result of the comparison is either "true" (-1) or "false" (0). You may use this result to control the flow of a program. (See the description of the IF statement.)

Table 2-2 summarizes the relational operators.

Table 2-2. Vectra BASIC Relational Operators

Operator	Relation	Sample Expression
=	Equality	COUNTER = LIMIT
<>	Inequality	LENGTH <> HEIGHT
<	Less than	COLUMN < 80
>	Greater than	ROW > 24
<=	Less than or equal to	YEAR <= 1984
>=	Greater than or equal to	LINES >= PAGE

You may also use the equal sign to assign a value to a variable. (See the description of the LET statement.)

When arithmetic and relational operators are combined in one expression, Vectra BASIC performs all arithmetic operations first. For example, the expression:

TMARGIN + BMARGIN \* LINECOUNT <= PAGESIZE/2

is true when the sum of TMARGIN, BMARGIN, and LINECOUNT is less than or equal to half the PAGESIZE.

## Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result that is either true (not zero) or false (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of the logical operators are summarized in the following truth tables. The operators are listed in their order of precedence.

### NOT

**Purpose:** NOT inverts its operand. That is, a true bit is set to false and a false bit is set to true.

**Truth Table:**

X	NOT X
1	0
0	1

### AND

**Purpose:** AND requires both operands to be true if the result is to be true.

**Truth Table:**

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

### OR – Inclusive OR

**Purpose:** OR returns true when either operand or both operands are true.

**Truth Table:**

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

### XOR – Exclusive OR

**Purpose:** XOR returns true when either operand is true.

**Truth Table:**

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

### IMP – Implied

**Purpose:** IMP returns true when both operands are the same. If they differ, the result is the same as the second operand.

**Truth Table:**

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

### EQV – Equivalent

**Purpose:** EQV returns true when both operands have the same value.

**Truth Table:**

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a value that determines program flow. For example,

```
IF VALUE < 0 OR VALUE > 100 THEN 480
IF QUARTER < 4 AND YEAR = 1983 GOTO 1000
IF NOT LIMIT THEN 100
```

Logical operators convert their operands to sixteen bit, signed, two's-complement integers in the range  $-32768$  to  $+32767$ . (If either operand is outside this range, an error occurs.) When both operands are given as 0 or  $-1$ , logical operators return 0 or  $-1$ . The given operation is performed on these integers in bitwise fashion, that is, each bit of the result is determined by the corresponding bits in the two operands.

You may use logical operators to test bytes for a particular bit pattern. For instance, you may use the AND operator to **mask** all but one of the bits of a status byte. Similarly, you may use the OR operator to merge two bytes to create a particular binary value.

The following examples demonstrate how you may use the logical operators in this fashion. (Each number is represented in two bytes, or 16 bits; however, the examples ignore any leading zeros.)

Operation	Calculation
63 AND 16 = 16	63 is binary 111111 and 16 is binary 10000 so 111111 AND 10000 is 10000 (or 16).
15 AND 14 = 14	15 is binary 1111 and 14 is binary 1110 so 1111 AND 1110 is 1110 (or 14).
-1 AND 8 = 8	$-1$ is binary 1111111111111111 and 8 is binary 1000 so 1111111111111111 AND 1000 is 1000 (or 8).
4 OR 2 = 6	4 is binary 100 and 2 is binary 10 so 100 OR 10 is 110 (or 6).
10 OR 10 = 10	10 is binary 1010, so 1010 OR 1010 is 1010 (or 10).
-1 OR -2 = -1	$-1$ is binary 1111111111111111 and $-2$ is binary 1111111111111110 so 1111111111111111 OR 1111111111111110 is 1111111111111111 (or $-1$ ).
TWOComp = (NOT X)*1	The two's-complement of any integer is the bit complement plus one. For example, if $x$ is equal to 2, NOT $x$ would be binary 1111111111111101. This is decimal $-3$ , and $-3$ plus 1 is $-2$ , or the complement of 2.

## Functional Operators

A function is a predetermined operation that performs the specified task on its operand. Vectra BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine).

Vectra BASIC also allows "user-defined" functions that you write. See the DEF FN statement for further details.

## String Operations

Vectra BASIC provides two string operations. These operations are string concatenation and string comparisons. (See the section on "String Functions" in Chapter 5 for a listing of the built-in functions that manipulate strings.)

**Concatenation** You can join strings together (concatenate them) by using the plus sign (+). For example,

```
10 A$ = "File" : B$ = "Name"
20 PRINT A$ + B$
30 PRINT "Another " + A$ + B$
RUN
FileName
Another FileName
```

**Comparisons** You can compare strings by using the same relational operators that you use for numeric comparisons:

```
= <> < > <= >=
```

Vectra BASIC compares strings by taking one character at a time from each string and comparing their ASCII codes. When all the ASCII codes are the same, the strings are equal. When the ASCII codes differ, the lower code number precedes the higher number. If, during a string comparison, Vectra BASIC reaches the end of one string while characters still remain in the other, the shorter string is said to be smaller. Leading and trailing blanks are significant. For example,

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"FILENAME" <> "filename"
"lg" > "kg"
"123" > "123"
"SMYTH" < "SMYTHE"
B$ < "52/4/24" (where B$ = "47/5/10")
```

You may use string comparisons to test string values or to alphabetize strings. When using string constants in comparison expressions, you must enclose the constant in quotation marks.

**Chapter 3:**

**The Vectra BASIC Environment**

- 3-1** Introduction
- 3-1** Vectra BASIC
- 3-7** Redirecting Input and Output

**3**

**The Vectra BASIC Environment**

**Introduction**

Chapter 1 describes the easiest procedure for running Vectra BASIC on your computer. However, entering Vectra BASIC through an MS-DOS system command gives you added flexibility in establishing the Vectra BASIC environment.

The first part of this chapter describes GMBASIC, the MS-DOS command that you must use to enter Vectra BASIC. The last part gives further information on redirecting input and output.

**Chapter**

**Vectra BASIC**

**Format:**

GMBASIC

```
GMBASIC [ [ <sidin> ] [ <sidout> ] ] [ filename ]  
[ /I ] [ /F, numfiles ] [ /S: reel ] [ /C: buffersize ]  
[ /M: highest mem.loc ] [ , max.block.size ] [ /D ]
```

**Purpose:**

Loads the Vectra BASIC interpreter program into your computer's memory.

**Remarks:**

<sidin> instructs Vectra BASIC to redirect input from the file named <sidin> (Input normally comes from the keyboard.) When you select this option, you must include it before you set any switches (for example, /C: or /M:.)

> *stdout* instructs Vectra BASIC to redirect output to the file named *stdout*. (Output normally goes to the computer's screen.) When you select this option, you must include it before you set any switches.

*filename* directs Vectra BASIC to run the specified Vectra BASIC program immediately. You may use this parameter to run programs in batch mode by including the filename in the command line of a .BAT file (such as AUTOEXEC.BAT). You must end each program with a SYSTEM statement. This allows the next command from the .BAT file to execute.

/I directs Vectra BASIC to statically allocate space for file operations that are based upon the /S and /F switches.

Normally, Vectra BASIC dynamically allocates space to support file operations. Thus, the /S and /F switches are usually unnecessary. Certain applications were written, however, that require internal data structures to be static. Under these circumstances, you should set the /I switch, then make the necessary settings with the /F or /S switches.

/F : sets the number of files that you can open simultaneously. However, Vectra BASIC ignores this switch unless you have set the /I switch. When both this switch and the /I switch are present, Vectra BASIC sets the number of files that may be opened simultaneously to the given number. Each file requires 62 bytes for the File Control Block (FCB), 128 bytes for the random-access data buffer, and 132 bytes for the standard Input/Output buffer. You may alter the size of the data buffer with the /S option switch. When you omit the /F parameter, Vectra BASIC sets the value to 5.

The number of open files that MS-DOS supports depends upon the value of the FILES= parameter in the CONFIG.SYS file. When you are using Vectra BASIC, we recommend that you set the FILES parameter to 10. Vectra BASIC allocates the first three files to Stdin, Stdout, Stderr, Sdaux, and Stdprn, then it sets aside an additional file for LOAD, SAVE, CHDIR, NAME, and MERGE commands. When you set FILES=10, six files remain for Vectra BASIC input/output files. Thus, /F:6 is the maximum number of files that you may request when FILES=10 appears in the CONFIG.SYS file.

Attempting to open a file after all the file handles have been taken results in a Too many files error message.

/S : sets the maximum record size. However, Vectra BASIC ignores this parameter unless you set the /I switch. When both this switch and the /I switch are present, Vectra BASIC sets the maximum record size for random-access files to *recL*.

When you omit this parameter, Vectra BASIC sets the value to 128 bytes.

### Note

The record size option for the OPEN statement cannot exceed this value.

/C : sets the buffer size for the RS-232 buffer as follows:

/C:0 disables RS-232 support. Any subsequent I/O operation results in a Device unavailable error.

/C:n allocates *n* bytes for the receive buffer and 128 bytes for the transmit buffer for each RS-232 port that is opened.

/C : when you omit the /C parameter, Vectra BASIC allocates 256 bytes for the receive buffer and 128 bytes for the transmit buffer.

*/M*: This switch is used when space is needed for machine language programs. Two parameters control how much space is allocated for Vectra BASIC's workspace and machine language programs, and where the space for machine language programs is allocated.

When this switch is not in use, Vectra BASIC allocates 64k (65536 bytes) of memory for your program workspace and no space is specially reserved for machine language programs.

The first parameter, *highest memory loc*, specifies the total amount of space that Vectra BASIC can use. This parameter must be an unsigned integer. Its highest value can be 65529; its lowest value must be greater than the amount taken by BASIC for its stack and file buffers; the lowest recommended value is 5000 bytes.

The second parameter, *max block size*, specifies the amount of space needed for Vectra BASIC's workspace PLUS the amount to be reserved for machine language programs. *max block size* must be specified in paragraphs (byte multiples of 16). If this parameter is not used, the value 4096 is assumed ( $4096 \times 16 = 65536$ ). This default value matches the default space allocated for BASIC workspace.

If the total space needed for your BASIC workspace and machine language routines is less than 64k, you need to use only the first parameter. For example, */M:32768* saves 32768 bytes for BASIC; the remaining 32768 bytes can be used by machine language programs.

If you need more space, the second parameter can reserve space outside the BASIC workspace. This parameter can be critically important if you are also using the SHELL command; it will prevent any shelled process from overwriting your machine language routines. */M: , 4196* saves 65536 bytes for Vectra BASIC and protects 1600 bytes outside the Vectra BASIC workspace for your machine language programs.

If you need to use SHELL with an extremely large program that won't fit into memory with when BASIC occupies its full 64k, the */M* switch can shrink the space that BASIC uses. */M: , 2048* allocates 32768 bytes for the Vectra BASIC workspace ( $2048 \times 16 = 32768$ ). */M: 32000 , 2048* allocates 32768 bytes, 32000 for the Vectra BASIC workspace and 768 bytes for machine language programs.

The following list shows how the */M* parameters affect memory allocation:

Specifications	Workspace	ASM Subroutines
default setting	65536	0
<i>/M: 49999</i>	50000	0
<i>/M: , 4000</i>	64000	0
<i>/M: 49999 , 4000</i>	50000	14000
<i>/M: , 4196</i>	65536	1600

*/D* retains the "Double Precision Transcendental" mathematical package in computer memory. When you omit this parameter, Vectra BASIC frees this area for program use.

### Examples:

The first example uses the default settings. Thus, it uses 64K of memory, permits 3 opened files, then loads and executes `PAYROLL.BAS`:

A> GWBASIC PAYROLL

The second example also uses 64K of memory but permits 6 opened files. It loads and executes `INVENT.BAS`:

A> GWBASIC INVENT / I:6

The next example disables RS-232 support and uses the first 32K bytes of memory. The memory above 32K is free for the user:

A> GWBASIC /C:0/M:32767

The next example statically allocates 4 file buffers and sets a maximum record length of 512 bytes:

A> GWBASIC /I/F:4/S:512

The last example uses 64K of memory and permits 3 opened files, allocates 512 bytes to the RS-232 receive buffers and 129 bytes to transmit buffers; it loads then executes `TTY.BAS`:

A> GWBASIC TTY/C:512

## Redirecting Input and Output

### Format:

GWBASIC *filename* [*< stdin*] [*> stdout*]

### Purpose:

You can instruct Vectra BASIC to read from a "standard input" file and write to a "standard output" file by providing the appropriate file names on the command line.

### Remarks:

Normally, Vectra BASIC receives its input from the keyboard. When you redirect input with the *stdin* parameter, Vectra BASIC reads from the specified file when it encounters any `INPUT`, `LINE INPUT`, `INPUT#`, or `INKEY$` statement.

Vectra BASIC continues to read from the source file until it detects a Control-Z. When it encounters a Control-Z, Vectra BASIC ends program execution and returns control to the operating system. If a file does not end with Control-Z or if a file input statement tries to read past the end-of-file, Vectra BASIC closes all open files, displays the message: `Read past end` to standard output, and returns control to MS-DOS.

When input is redirected, Vectra BASIC still responds to a `[CTRL] Break` from the keyboard. If the program requests input from the "KYBD:" device, this input is read from the keyboard, rather than *stdin*. Furthermore, Vectra BASIC continues to trap keys from the keyboard when the `ON KEY(n)` statement appears within a program.

Normally, Vectra BASIC writes to the computer's screen. When you redirect output with the *stdout* parameter, Vectra BASIC writes to the specified file (and not to the screen) when it encounters any `PRINT` statements. If the program sends output to "SCRN:" as a device, this information is printed on the screen, not in the *stdout* file.

If input has not been redirected, output and error messages are sent to both the screen and the standard output file.

Pressing **CTRL** and **BREAK** causes Vectra BASIC to close all open files except `stdout`. The message `Break in line number` is written to the screen and to `stdout`. All subsequent screen output is written to the `stdout` file as well as to the screen. To end the redirection of output, you must issue a `SYSTEM` command, which closes the `stdout` file when it returns you to DOS.

### Caution

You should not attempt to `KILL` or rename a file that is being used at `stdout`.

### Examples:

The first example reads data for `INPUT` and `LINE INPUT` statements from the keyboard. `PRINT` statements write data to the file `DATA.OUT`:

```
A> GWBASIC MYPROG >DATA.OUT
```

The next example reads data for `INPUT` and `LINE INPUT` statements from the file `DATA.IN`. Any `PRINT` statements send data to the screen:

```
A> GWBASIC MYPROG <DATA.IN
```

The third example reads data for `INPUT` and `LINE INPUT` statements from the file `DATA.IN`. Any `PRINT` statements write to the file `MYOUTPUT.DAT`:

```
A> GWBASIC MYPROG <MYINPUT.DAT  
>MYOUTPUT.DAT
```

The last example reads data for `INPUT` and `LINE INPUT` statements from the file identified by the pathname `\SALES\JOHN\TRANS`. All `PRINT` statements append data to the file `\SALES\SALES.DAT`:

```
A> GWBASIC MYPROG <\SALES\JOHN\TRANS  
>> \SALES\SALES.DAT
```

### Note

If you only included one output redirection symbol (`>`), Vectra BASIC would overwrite the file instead of adding to it.

## Chapter 4:

### Directory and File Operations

- 4-1 Directory Paths
- 4-2 Disc File Naming Conventions
- 4-3 Disc Filenames
- 4-3 Disc Data Files
- 4-3 Sequential Files
- 4-4 Random Files
- 4-5 Creating a Random File
- 4-6 Accessing a Random File
- 4-9 Protected Files

# 4

## Directory and File Operations

### Directory Paths

Vetra BASIC runs under MS-DOS 3.1 (or later versions). This operating system allows tree-structured directories. That is, a directory can contain both files and other directories.

You may use one of two methods to specify a file name. You can either specify a path from the root (the base of the tree-structured hierarchy); or you can specify a path from the current directory (the directory in which you are currently working).

A path is a series of directory names that are separated by backslashes. A single backslash represents the root directory.

A typical path might appear as:

```
\USER1\CRAIG\GAMES\CHES\WDM
```

The notation `..` refers to the directory directly above the current directory. This directory is called the parent directory. Regardless of your current directory, you may always refer to that directory's parent by typing two periods (`..`).

### Note

The root is its own parent.

No restriction exists on the depth of a tree. (The **depth** is defined as the longest path from the root to an end directory (or leaf)). However, the root directory has a fixed number of entries. These are the limits for root directories:

Type of disc	Capacity	Directory Entries
Flexible	160 Kbytes	66
Flexible	320/360 Kbytes	112
Flexible	1.2 Mbytes	224
Hard	20 Mbytes	512

The other directories have no limit to the number of files they may contain as long as space remains available.

## Disc File Naming Conventions

All Vectra BASIC instructions that use filenames now allow a directory path to be included, as well as the drive designator. The order is always drive, path, filename. These instructions are **BLOAD**, **BSAVE**, **CHAIN**, **FILES**, **KILL**, **LOAD**, **MERGE**, **NAME**, **OPEN**, **RUN**, and **SAVE**.

The drive designator consists of one letter and a colon.

A pathname may not exceed 128 characters. Pathnames longer than 128 characters give a **Bad filename** error.

When you include the drive designator in a pathname, you must list it first. For example,

**B:** \TRAVELS\JENNEFER\ISRAEL is legal, while  
 \TRAVELS\JENNEFER\B: ISRAEL is not.

Placing a drive designator other than at the beginning of the pathname, results in a **Bad filename** error.

When you omit the drive designator, Vectra BASIC assumes that you are referring to the currently active disc.

## Disc Filenames

Disc filenames obey the standard MS-DOS naming conventions. (Refer to your Owner's Guide.) All filenames may include a letter and a colon as the first two characters to specify a disc drive. For example, **A:** refers to drive A. If you omit this special symbol combination, Vectra BASIC assumes that all files refer to the currently selected disc drive. When you use either the **LOAD**, **SAVE**, **MERGE**, or **RUN** statements, Vectra BASIC attaches the file type extension **.BAS** to the filename if you omit a file extension.

## Disc Data Files

You may create two different types of disc data files for a Vectra BASIC program to access. They are sequential files and random access files.

## Sequential Files

Sequential files have a simpler structure than random-access files, but they are limited in their flexibility and their speed of accessing data. When you write data to a sequential file, Vectra BASIC writes the information to the file in sequential order, one item after the other, in the order that it is sent. Vectra BASIC reads back the information in the same way.

You may use the following statements and functions with sequential files:

```
CLOSE
EOF
INPUT#
LINE INPUT#
LOC
LOF
OPEN
PRINT# USING
WRITE#
```

You must follow these steps to create a sequential file, then access its data:

1. Open the file in **O** mode. For example,  

```
OPEN "Q", #1, "DATA"
```
2. Write data to the file using the **PRINT#** or **WRITE#** statement. For example,  

```
WRITE #1, A$;B$;C$
```
3. To access the data in the file, you must close the file then reopen it in **R** mode. For example,  

```
CLOSE #1  
OPEN "1", #1, "DATA"
```
4. Use the **INPUT#** statement to read data from the sequential file into the program. For example,  

```
INPUT #1, X$,Y$,Z$
```

A program that creates a sequential file can also write formatted data to the disc with the **PRINT# USING** statement. For example, you could use the following statement to write numeric data to disc without using explicit delimiters:

```
PRINT #1, USING "###.##", A, B, C, D
```

In this example, the comma at the end of the format string (before the closing quotation mark) separates the items in the disc file.

## Random Files

It takes more programming steps to create and access random files than sequential files. However, you may find the advantages of random-access files outweigh the time required to enter the extra steps.

With random files, Vectra BASIC stores and accesses information in distinct units called **records**. Since each record is numbered, you may access data anywhere in the file without reading through the file sequentially.

You may use the following statements with random-access files:

```
CLOSE  
CVI CVS CVD  
FIELD  
GET  
LOC  
LOF  
LSET/RSET  
MKI$ MKS$ MKD$  
OPEN  
PUT
```

**Creating a Random File** You must follow these steps to create a random file:

1. Open the file for random access (**R** mode). The following example sets a record length of 32 bytes. When you omit the record length parameter, Vectra BASIC uses 128 bytes as the default record size. (To change the default size, see the discussion of **/S** in Chapter 3.)

```
OPEN "R", #1, "FILE", 32
```

## Note

The maximum logical record number is 32767. Theoretically, if you set the record size to 256 bytes, you may access files up to 8 megabytes in size.

2. Use the **FIELD** statement to allocate space in the random file buffer for the variables that you plan to write to the random file. For example,  

```
FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
```
3. Use **LSET** to move the data into the random file buffer. However, before you place numeric values into this buffer, you must convert these values to strings by using one of the following functions:

```
MKI$ Converts an integer value to a string  
MKS$ Converts a single-precision value to a string  
MKD$ Converts a double-precision value to a string
```

Examples of the LSET statement are:

```
LSET N$ = X$
LSET A$ = MK$(AMT)
LSET P$ = TEL$
```

4. Write the data from the buffer to the disc using the PUT statement:  

```
PUT #1, CODEX
```

The following example creates a random access file. If you use it to create several records, you will be able to access them with the program in the next section. You can end the program by pressing **Enter** without typing a name.

```
10 CODEX=10
20 OPEN "R", #1, "FILE", 32
30 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
40 INPUT "NAME: ", X$
50 IF X$="" THEN CLOSE #1:END
60 INPUT "AMOUNT PLEDGED: ", AMT$
70 INPUT "TELEPHONE NUMBER: ", TEL$
80 LSET N$ = X$
90 LSET A$ = MK$(AMT)
100 LSET P$ = TEL$
110 PUT #1, CODEX
120 CODEX=CODEX+1:GOTO 40
```

**Accessing a Random File** You must follow these steps to access the data in a random-access file:

1. Open the file for random access (R mode). For example,  

```
OPEN "R", #1, "FILE", 32
```
2. Use the FIELD statement to allocate space in the random file buffer for the variables that you plan to read from the file. For example,  

```
FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
```

### Note

In a program that performs both input and output on the same random file, you can usually use one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random file buffer. In the following example, CODEX contains the record number.

```
GET #1, CODEX
```

4. Your program may now access the data in the buffer. However, numeric values must be converted from strings back to numbers. You do this with the convert functions:

```
CVI Converts the data item to an integer
CVS Converts the data item to a single-precision value
CVD Converts the data item to a double-precision value
```

For example:

```
PRINT CVS(A$)
```

In the following example, the user accesses the random file called FILE by entering a 2-digit code at the keyboard. The program then reads the information that is associated with the code and displays it on the computer screen

```
10 OPEN "R", #1, "FILE", 32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE: ", CODEX
40 IF CODEX = 0 THEN CLOSE 1 : END
50 GET #1, CODEX
60 PRINT N$
70 PRINT USING "#####.###"; CVS(A$)
80 PRINT P$ : PRINT
90 GOTO 30
```

The following program illustrates random file access. In this program, the record number serves as the part number. (It is assumed that the inventory never contains more than 100 different part numbers.) Lines 900 through 960 initialize the data file by writing CHR\$(255) as the first character of each record. Later, lines 270 and 500 use this character to determine whether an entry already exists for that part number

```
110 OPEN "R", #1, "INVEN.DAT", 39
120 FIELD #1, 1 AS F$, 30 AS D$, 2 AS O$,
2 AS R$, 4 AS P$
130 PRINT : PRINT "FUNCTIONS: " : PRINT
140 PRINT 1, "INITIALIZE FILE"
```

```

150 PRINT 2, "CREATE NEW ENTRY"
160 PRINT 3, "DISPLAY INVENTORY FOR ONE PART"
170 PRINT 4, "ADD TO STOCK"
180 PRINT 5, "SUBTRACT FROM STOCK"
190 PRINT 6, "DISPLAY ALL ITEMS
    BELOW REORDER LEVEL"
200 PRINT 7, "END PROGRAM"
210 PRINT : PRINT : INPUT "FUNCTION"; FUNCTION
220 IF (FUNCTION < 1) OR (FUNCTION > 7)
    THEN PRINT "BAD FUNCTION NUMBER" : GOTO 130
230 ON FUNCTION GOSUB 900,250,390,460,560,660,860
240 GOTO 130
250 REM BUILD NEW ENTRY
260 GOSUB B40
270 IF ASC(F$) <> 255 THEN INPUT "OVERWRITE"; A$:
    IF A$ <> "Y" THEN RETURN
280 LSET F$ = CHR$(0)
290 INPUT "DESCRIPTION", DESC$
300 LSET D$ = DESC$
310 INPUT "QUANTITY IN STOCK", Q$
320 LSET Q$ = MKI$(Q$)
330 INPUT "REORDER LEVEL", R$
340 LSET R$ = MKI$(R$)
350 INPUT "UNIT PRICE"; P
360 LSET P$ = MK$(P)
370 PUT #1, PART$
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB B40
410 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY" : RETURN
420 PRINT USING "PART NUMBER ###"; PART$
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND ###"; CVI(Q$)
450 PRINT USING "REORDER LEVEL ###"; CVI(R$)
460 PRINT USING "UNIT PRICE ###.##"; CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB B40
500 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY" : RETURN
510 PRINT Q$ : INPUT "QUANTITY TO ADD ", A$
520 Q$ = CVI(Q$) + A$
530 LSET Q$ = MKI$(Q$)
540 PUT #1, PART$
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB B40
580 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY" : RET
590 PRINT Q$
600 INPUT "QUANTITY TO SUBTRACT"; S$
610 Q$ = CVI(Q$)
620 IF (Q$ - S$) < 0 THEN PRINT "ONLY"; Q$;
    " IN STOCK" : GOTO 600
630 Q$ = Q$ - S$
640 IF Q$ < CVI(R$)
    THEN PRINT "QUANTITY NOW"; Q$;
    "REORDER LEVEL"; CVI(R$)
    "REORDER LEVEL"; CVI(Q$)
650 LSET Q$ = MKI$(Q$)
660 PUT #1, PART$
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I = 1 TO 100
700 GET #1, I
710 IF ASC(F$) = 255 THEN GOTO 730
720 IF CVI(Q$) < CVI(R$) THEN PRINT Q$; "QUANTITY"
    CVI(Q$) TAB(50) "REORDER LEVEL"; CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER"; PART$
850 IF (PART$ < 1) OR (PART$ > 100)
    THEN PRINT "BAD PART NUMBER" : GOTO 840
    ELSE GET #1, PART$ : RETURN
860 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE"; B$:
    IF B$ <> "Y" THEN RETURN
920 LSET F$ = CHR$(255)
930 FOR I = 1 TO 100
940 PUT #1, I
950 NEXT I
960 RETURN

```

## Protected Files

If you wish to save a program in a special binary format, you must use the "Protect" (P) option with the SAVE command. For example, the following statement saves the program named ETERNAL so it cannot be listed or edited:

```
SAVE "ETERNAL", P
```

As no command exists to "unprotect" the file, you may also want to save an unprotected copy of the program that you can list and change.

## Chapter 5:

### Programming Tasks

5-1	Introduction
5-3	System Commands
5-5	Using Commands as Program Statements
5-6	Directory Operations
5-6	File Operations
5-8	Defining and Altering Data and Variables
5-9	Computer Control
5-10	Graphics
5-12	Music
5-13	Program Control, Branching, and Subroutines
5-16	Terminal Input and Output
5-17	RS-232 Asynchronous Communications
5-18	Event Trapping
5-19	Communications Trapping
5-19	Key Trapping
5-19	Pen Trapping
5-20	Play Trapping
5-20	Joystick Trapping
5-20	Timer Trapping
5-21	Error Trapping
5-22	Debugging Aids
5-23	Vectra BASIC Functions
5-24	General Purpose Functions
5-24	Device Sampling Functions
5-25	Input/Output Functions
5-26	Arithmetic Functions
5-27	Derived Functions
5-29	String Functions
5-30	Special Functions

# 5

## Programming Tasks

### Introduction

When programming, you normally have a specific task that you wish to perform. The experienced programmer has no difficulty determining which Vectra BASIC instruction is appropriate for the task at hand. However, if some features of the language are new to you, you may have trouble isolating the best instruction. This chapter groups the various Vectra BASIC commands, statements, and functions into task-oriented areas. For example, if you want to draw a figure, you may know that you need a "graphics" statement, but you may not know which one. By looking under the graphics section in this chapter, you would discover that Vectra BASIC provides three "drawing" statements: **DRAW**, **LINE**, and **CIRCLE**. You can get an indication of each statement's use by reading its general description. Then you should consult Chapter 6 for full details on using the statement that you selected.

## Chapter 5

This chapter contains the following sections:

- System commands
- Using system commands as program statements
- Directory operations
- File operations
- Defining and altering data and variables
- Computer control
- Graphics
- Music
- Program control, branching, and subroutines
- Terminal input and output
- RS-232 asynchronous communications
- Event trapping
- Error trapping
- Debugging aids
- General purpose functions
- Input/Output functions
- Device sampling functions
- Arithmetic functions
- Derived arithmetic functions
- String functions
- Special functions

## System Commands

System Commands are those commands that you enter on the Vectra BASIC command line and/or those that return control to the command line. The following list summarizes the system commands that Vectra BASIC provides.

<b>AUTO</b>	Automatically generates line numbers for program entry.
<b>BL0AD</b>	Loads the specified memory image file into your computer's memory.
<b>BSAVE</b>	Saves the contents of the specified area of memory to a disc file.
<b>CONT</b>	Continues program execution after you type a Control-C, or your program executes a STOP or END statement.
<b>DELETE</b>	Removes the specified lines from a Vectra BASIC program.
<b>EDIT</b>	Displays a line for editing.
<b>ENVIRON</b>	Modifies parameters in the Vectra BASIC environment table, which is used with the SHELL command.
<b>FILES</b>	Lists the names of the files residing on a specified disc.
<b>KILL</b>	Deletes one or more files from a specified disc.
<b>LIST and LLIST</b>	Lists all or part of the program that is currently stored in memory to either the computer screen or a printer.

LOAD	Loads a Vectra BASIC program file from disc into memory.
MERGE	Incorporates statements contained in the specified disc file into the program that is currently stored in computer memory.
NEW	Deletes the program that is currently stored in computer memory and clears all variables.
RENUM	Renumbers the lines of a program so they occur in a specified sequence.
RESET	Closes all disc files and prints the directory information to every disc with open files.
RUN	Executes the program that is currently stored in your computer's memory or on a disc file.
SAVE	Saves the program currently stored in computer memory to a specified disc file.
SHELL	Branches from the Vectra BASIC interpreter to run a .COM, .EXE or BAT program, or a DOS function.
SYSTEM	Exits Vectra BASIC and returns control to the operating system.

## Using Commands as Program Statements

You may use several of the Vectra BASIC commands as program statements.

Refer to the preceding discussion for each of the commands, then consult this section for its use within a program.

BLOAD	Programmatically loads code or data into a given area of memory.
BSAVE	Programmatically copies code or data from memory to a specified disc file.
FILES	Programmatically lists directory information.
KILL	Programmatically deletes the specified disc files.
RESET	Programmatically closes all disc files and prints the directory information to every disc with open files. (You should use this statement in any program that performs disc access.)
RUN	Programmatically re-executes a program from a specified line, or loads and executes a new program.
SHELL	Branches to DOS command level to run a .COM, .EXE, or .BAT program, or a DOS function.
SYSTEM	Programmatically exits Vectra BASIC.

## Directory Operations

The following list summarizes those statements that have been added to Vectra BASIC to handle directories.

CHDIR	Changes the current directory.
MKDIR	Creates a directory on the specified disc.
RMDIR	Removes a directory from the specified disc.

## File Operations

Vectra BASIC provides the following instructions for handling files and their contents.

CLOSE	Concludes all input/output to a disc file.
EOD	Returns end-of-file status for sequential and random-access files.
FIELD	Allocates space for variables in a random file buffer.
GET	Reads a record from a random disc file into a random file buffer.

INPUT#	Reads values from a sequential disc file and assigns them to program variables.
LINE INPUT#	Reads an entire line (up to 254 characters) from a sequential disc file and assigns the line to a string variable.
LOC	Returns current record number of a file, as determined by the last GET or PUT statement.
LOF	Returns the length of the file, in bytes.
LSET and RSET	Moves data from memory into random file buffer variables in preparation for a PUT statement.
NAME	Changes the name of a disc file.
OPEN	Allows access to a file for reading and/or writing.
PRINT# and PRINT# USING	Writes data to a sequential disc file.
PUT	Writes a record from a random file buffer to a random disc file.
WIDTH	Sets the printer line width by specifying the number of characters per line.
WRITE#	Writes data to a sequential file.

## Defining and Altering Data and Variables

Vectra BASIC provides several statements that you may use within a program to define and manipulate data, variables, expressions, and arrays. The following list summarizes these statements.

<b>CLEAR</b>	Sets numeric and string variables to zero or null, closes all files, and optionally sets the end of memory and the amount of stack space.
<b>COMMON</b>	Passes variable values to a chained program.
<b>DATA</b>	Stores data for later access by a program's READ statements.
<b>DEFINT/DEFSGN DEFDBL/DEFSTR</b>	Declares that Vectra BASIC should automatically treat certain variable names as integer, single-precision, double-precision, or string variable types.
<b>DIM</b>	Sets the maximum values for an array's subscripts, allocates storage, and assigns an initial value to array elements.
<b>ERASE</b>	Removes an array from a program.
<b>LET</b>	Assigns the value of an expression to a variable.
<b>MID\$</b>	Replaces a portion of one string with another string.
<b>OPTION BASE</b>	Determines if the minimum value for an array subscript should be zero or one.
<b>READ</b>	Reads values from a DATA statement and assigns them to variables.

## Computer Control

Vectra BASIC provides several statements that interface with your computer. The following list summarizes these statements.

<b>BEEP</b>	Sounds the computer's bell.
<b>CLS</b>	Erases the specified contents from the display screen.
<b>COLOR</b>	Selects both foreground and background color and also character enhancements.
<b>DATE\$</b>	Sets the system date.
<b>INP</b>	Returns a byte which is read from a microprocessor port.
<b>KEY</b>	Assigns user-defined expressions to the function keys.
<b>LOCATE</b>	Moves the alphanumeric cursor to the specified screen location.
<b>OUT</b>	Sends a byte to a microprocessor port.
<b>PEEK</b>	Reads a byte from a memory location.
<b>POKE</b>	Writes a byte into a memory location.
<b>TIME\$</b>	Sets the system time.
<b>WAIT</b>	Suspends program execution while monitoring the status of a microprocessor input port.

## Graphics

Vetra BASIC's extended graphics statements add significant capabilities to BASIC. Powerful statements such as **POINT**, **LINE**, and **CIRCLE** easily draw different shapes and figures. The **DRAW** statement and the graphics statements **GET** and **PUT** allow animation. The **VIEW** statement scales objects by placing them in small or large viewports. The **WINDOW** statement creates special effects such as "zoom" and "pan". **PMAP** maps pixel coordinates to Cartesian coordinates (and vice versa). You use these coordinates to scale the viewports.

The following list summarizes the graphics instructions that Vetra BASIC provides.

<b>CIRCLE</b>	Draws an ellipse on the screen.
<b>CLS</b>	Erases graphics images from the display screen.
<b>COLOR</b>	Selects the colors for the background, foreground and border colors in text mode; selects the background color and palette in medium resolution graphics, and selects the foreground color in high resolution graphics.
<b>DRAW</b>	Draws the specified object.
<b>GET</b>	Transfers graphics images from the screen into an array.
<b>LINE</b>	Draws a line or box on the screen.

<b>PAINT</b>	Fills an area on the screen with the selected color or tile pattern.
<b>PMAP</b>	Maps physical coordinates to "world" coordinates and vice versa.
<b>POINT</b>	Reads the attribute value of a given pixel.
<b>PRESET</b>	Changes the color attribute of a given pixel.
<b>PSET</b>	Draws a pixel at the specified coordinate with the given attribute.
<b>PUT</b>	Transfers an image from the specified array to the screen.
<b>SCREEN</b>	Changes between the text screen, the medium resolution graphics/text screen, and the high resolution graphics/text screen.
<b>VARPTR</b>	Returns the character form for the memory address of a variable.
<b>VIEW</b>	Defines subsets of the screen for graphics displays.
<b>WINDOW</b>	Redefines the screen coordinates.

## Music

The Vectra plays notes through the computer's speaker. The **SOUND** statement makes single sounds of a specified frequency and duration. The **PLAY** statement can play whole tunes, specified by note and octave. When used with the **ON PLAY** event-trapping statement, music can be continuously played in the background while Vectra BASIC executes other commands.

<b>ON PLAY</b>	Specifies where to branch when the Music Background buffer has only a specified number of notes remaining.
<b>PLAY</b>	Plays the specified string of notes, in the foreground or background.
<b>PLAY (n)</b>	Returns the number of notes remaining in the Music Background buffer.
<b>SOUND</b>	Plays a note of a specified duration and frequency.

## Program Control, Branching, and Subroutines

Vectra BASIC provides several statements that control the flow of program execution. This includes branching to other lines, subroutines, and programs. The following list summarizes these statements.

<b>CALL</b>	Calls an assembly-language subroutine.
<b>CALLS</b>	Calls an assembly-language subroutine with segmented addresses.
<b>CHAIN</b>	Calls a program and passes variable values to it from the current program.
<b>DEF FN</b>	Names and defines a user-written function.
<b>DEF SEG</b>	Assigns the current segment address. Subsequent <b>CALL</b> , <b>CALLS</b> , <b>POKE</b> , <b>PEEK</b> , or <b>USR</b> instructions refer to this address.
<b>DEF USR</b>	Assigns the starting address of an assembly-language subroutine.
<b>END</b>	Ends program execution, closes all files, and returns control to the command level.
<b>FOR . . . NEXT</b>	Loops through a series of instructions a given number of times.

<b>GOSUB . . . RETURN</b>	Branches to and returns from a subroutine.
<b>GO TO</b>	Branches unconditionally to the specified line number.
<b>IF</b>	Determines program flow based on the result returned by a logical expression.
<b>ON ERROR GOTO</b>	Enables error trapping and specifies the first line number of the error-handling subroutine.
<b>ON . . . GOSUB</b>	Branches to one of several specified subroutines, depending upon the value returned by the governing expression.
<b>ON . . . GOTO</b>	Branches to one of several specified line numbers, depending upon the value returned by the governing expression.
<b>RESUME</b>	Continues program execution after Vectra BASIC has performed an error recovery procedure.
<b>RETURN</b>	Returns control to the statement following the last-executed <b>GOSUB</b> statement.
<b>STOP</b>	Suspends program execution and returns control to the Vectra BASIC command line.
<b>WHILE . . . WEND</b>	Loops through a series of statements as long as a given condition is true.

You may divide the branching and subroutine statements into the following categories:

Unconditional branching:  
GOTO

Conditional branching:  
IF . . . THEN (. . . ELSE)  
IF . . . GOTO  
ON ERROR GOTO  
ON . . . GOTO  
WHILE . . . WEND

Branching to another program:  
CHAIN

Looping:  
FOR . . . NEXT  
WHILE . . . WEND

Subroutines:  
CALL  
CALL S  
DEF FN  
DEF SEG  
DEF USR  
GOSUB . . . RETURN  
ON . . . GOSUB  
RETURN

## Terminal Input and Output

You may use Vectra BASIC input statements for entering information into programs from either the keyboard, disc files, or the DATA statement. You may use Vectra BASIC output statements to copy information to the computer screen, a printer, a file, and/or a memory location. The following list summarizes these statements.

INPUT	Takes input from the keyboard.
IOCTL	Prints a control character or string to a device driver.
IOCTL\$	Reads a control character string from a device driver.
LINE INPUT	Enters an entire line (up to 254 characters) to a string variable, without the use of delimiters.
LPRINT and LPRINT USING	Prints data to a line printer.
PRINT	Prints data to the computer screen.
PRINT USING	Uses a specified format to print strings or numbers.
VIEW PRINT	Creates a text window on the screen.
WIDTH	Sets the printer line width by specifying the number of characters per line.
WRITE	Writes data to the computer screen.

## RS-232 Asynchronous Communications

The RS-232 Asynchronous Communication option permits Vectra BASIC to "talk" with other computers and peripherals. For example, this capability provides printer and plotter support.

EOF	Tells whether the input queue is empty.
GET/PUT	Permit fixed-length I/O for communication.
INPUT\$	Returns a string of characters from the specified file.
LOC	Returns the number of characters in the input queue that are ready to be read.
LOF	Returns the amount of free space in the input queue.
OPEN "COM	Opens a communications file by allocating a buffer for I/O.

## Event Trapping

Event trapping allows a program to transfer control to a specific program line if a certain event happens. After servicing the event, the trap routine executes a RETURN statement that returns program control to the place where the trap occurred.

Trapped events might include the receipt of characters from a communication port or the pressing of a function key.

You control event trapping through the following statements:

- event-specifier* ON Turns on trapping
- event-specifier* OFF Turns off trapping
- event-specifier* STOP Suspends trapping temporarily

*event-specifier* may be one of the following:

COM(n)  
KEY(n)  
PEN  
PLAY  
STRING(n)  
TIMER

### Note

All of these statements have an *event-specifier* ON format and an ON *event-specifier* format. Each form serves a specific purpose. For example, KEY(n) ON sets an event trap for the specified key, while ON KEY gives the line number where program control branches when the trap occurs

The following list summarizes the instructions that Vectra BASIC provides for event trapping. Each of the event traps uses a GOSUB. The RETURN statement resumes program execution at the point where the event took place.

RETURN Returns from an event-trapping routine.

### Communications

COM(n) Activates, deactivates, or suspends trapping of communications activity on the specified channel.

ON COM Specifies where to branch when activity occurs on a communications channel.

### Key Trapping

KEY(n) Activates, deactivates, or suspends trapping of the specified key.

ON KEY Specifies where to branch when the user presses the specified key.

### Pen Trapping

PEN Activates, deactivates, or suspends trapping of the light pen.

ON PEN Specifies where to branch when the light pen is used.

## Play Trapping

PLAY

Activates, deactivates, or suspends trapping of the Music Background buffer status.

ON PLAY

Specifies where to branch when the Music Background has the specified number of remaining notes.

## Joystick Trigger Trapping

STRIG(n)

Activates, deactivates, or suspends trapping of the specified joystick trigger.

ON STRIG

Specifies where to branch when the user presses the specified joystick trigger.

## Timer Trapping

TIMER

Enables, disables, or suspends TIMER event trapping.

ON TIMER

Specifies the time setting and the branch location for TIMER events.

## Error Trapping

Error trapping is a specialized type of event trapping. Usually, errors in Vectra BASIC cause an error message to be printed on the screen, and program execution stops. Error trapping allows a program to transfer control to error-handling subroutines, which can take specific actions based on the type of error which caused the trap.

ON ERROR GOTO

Enables error trapping and specifies the first line number of the error-handling subroutine.

RESUME

Continues program execution after Vectra BASIC has performed an error recovery procedure.

ERR and ERL

Returns the error number and line number for the last-encountered program error.

ERDEV

Returns the error code from the last device to declare an error.

ERDEV#

Contains the name of the device driver that generated the ERDEV error.

## Debugging Aids

You may use debugging statements to trace program execution, to define error codes, or to simulate error conditions. Since well-documented programs help prevent errors, we treat the **REM** statement as a debugging aid.

The following list summarizes the debugging statements that Vectra BASIC provides.

<b>CONT</b>	Continues execution of a program after a <b>[CTRL] [Break]</b> , <b>STOP</b> or <b>END</b> statement has halted program execution.
<b>ERROR</b>	Simulates the occurrence of a Vectra BASIC error, or allows you to define error codes.
<b>REM</b>	Inserts explanatory remarks into a program.
<b>STOP</b>	Ends program execution.
<b>TRON/TROFF</b>	Trace the execution of program statements.

## Vectra BASIC Functions

Vectra BASIC provides several intrinsic functions. You may call these functions, without further definition, from any point in a program.

You must enclose a function's argument(s) in parentheses. Most function formats abbreviate the arguments as follows:

*x* and *y*    Represent numeric expressions  
*i* and *j*    Represent integer expressions  
*x\$* and *y\$*    Represent string expressions

If you give a function a floating point value when the function takes an integer argument, Vectra BASIC rounds the fractional portion and uses the integer result.

You may divide the functions into five general categories. These categories are:

- General Purpose Functions
- Input/Output Functions
- Arithmetic Functions
- String Functions
- Special Functions

**General Purpose Functions.**

Vectra BASIC provides the following general-purpose functions:

- DATE\$** Returns the current system date.
- SCREEN** Returns either the ASCII code or a character attribute for the specified location.
- TIMER** Gives the number of seconds since midnight.
- TIME\$** Returns the current system time.

**Device Sampling Functions**

Vectra BASIC can read input from a joystick or lightpen. These functions return information to BASIC about the use of these devices:

- PEN(n)** Reads the light pen coordinates.
- STICK(n)** Reads the joystick coordinates.
- STRIG(n)** Reads presses of the joystick triggers.

Joystick and pen "events" can also be trapped by Vectra BASIC event trapping routines. See the section on event trapping for a list of these instructions.

**Input/Output Functions**

The Input/Output functions send or return information to the computer or a printer.

- CSRLIN** Returns the print head's row position.
- CV1, CV5, CVD** Convert string values to numeric values.
- EOF** Returns end-of-file status for sequential and random-access files.
- INKEY\$** Returns a one-character or null string from the computer's keyboard.
- INPUT\$** Returns a string from either the keyboard or a disc data file.
- LOC** Returns the current record number of a file, as determined by the last GET or PUT statement.
- LOF** Returns the length of the file, in bytes.
- LPDS** Returns the current position of the printer print head within the printer buffer.
- MK1\$, MK5\$, MKD\$** Convert numeric values to string values.
- PDS** Returns the print head's column position.
- SPC** Prints spaces (blank characters) on the display.
- TAB** Moves to a specified position on a line.

Arithmetic Functions

The RANDOMIZE statement and the arithmetic functions manipulate numeric expressions.

ABS	Returns the absolute value of the numeric expression.
ATN	Returns the arctangent of a numeric expression.
CDBL	Converts a numeric expression to a double-precision number.
CINT	Converts a numeric expression to an integer by rounding off the fractional part.
COS	Returns the cosine of a numeric expression which you must give in radians.
CSNG	Converts a numeric expression to a single-precision number.
EXP	Returns $e$ (where $e \approx 2.71828...$ ) to the power of $X$ . $X$ must be less than 88.02969.
FIX	Returns the truncated integer part of a numeric expression.

INT	Returns the largest integer value that is less than or equal to a given numeric expression.
LOG	Returns the natural logarithm of a numeric expression.
RANDOMIZE	Reseeds the random number generator.
RND	Returns a pseudo-random number between 0 and 1.
SGN	Returns 1 if a numeric expression is positive, returns 0 if the expression is equal to zero, and returns -1 if the expression is negative.
SIN	Returns the sine of a numeric expression which you must give in radians.
SQR	Returns the square root of a numeric expression.
TAN	Returns the tangent of a numeric expression which you must give in radians.

Derived Functions

Vectra BASIC provides intrinsic functions for your immediate use. From these intrinsic functions, you may derive the following functions:

Function	Equivalent
Secant	$SEC(X) = 1/COS(X)$
Cosecant	$CSC(X) = 1/SIN(X)$
Cotangent	$COT(X) = 1/TAN(X)$
Inverse Sine	$ARCSIN(X) = ATN(X/SQR(-X*X+1))$
Inverse Cosine	$ARCCOS(X) = -ATN(X/SQR(-X*X+1)) + 1.5708$

;

Function	Equivalent
Inverse Secant	$\text{ARCSEC}(X) = \text{ATH}(X/\text{SQR}(X^2 - 1)) + (\text{SGN}(X) - 1) \cdot 1.5708$
Inverse Cosecant	$\text{ARCCSC}(X) = \text{ATH}(X/\text{SQR}(X^2 - 1)) + (\text{SGN}(X) - 1) \cdot 1.5708$
Inverse Cotangent	$\text{ARCCOT}(X) = -\text{ATH}(X) + 1.5708$
Hyperbolic Sine	$\text{SINH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/2$
Hyperbolic Cosine	$\text{COSH}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$
Hyperbolic Tangent	$\text{TANH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/(\text{EXP}(X) + \text{EXP}(-X))$
Hyperbolic Secant	$\text{SECH}(X) = 2/(\text{EXP}(X) + \text{EXP}(-X))$
Hyperbolic Cosecant	$\text{CSCH}(X) = 2/(\text{EXP}(X) - \text{EXP}(-X))$
Hyperbolic Cotangent	$\text{COTH}(X) = \text{EXP}(X) + \text{EXP}(-X)/(\text{EXP}(X) - \text{EXP}(-X))$
Inverse Hyperbolic Sine	$\text{ARCSINH}(X) = \text{LOG}(X + \text{SQR}(X^2 + 1))$
Inverse Hyperbolic Cosine	$\text{ARCCOSH}(X) = \text{LOG}(X + \text{SQR}(X^2 - 1))$
Inverse Hyperbolic Tangent	$\text{ARCTANH}(X) = \text{LOG}((1+X)/(1-X))/2$
Inverse Hyperbolic Secant	$\text{ARCSCH}(X) = \text{LOG}((\text{SQR}(X^2 + 1) + 1)/X)$
Inverse Hyperbolic Cosecant	$\text{ARCCSCH}(X) = \text{LOG}((\text{SGH}(X) + \text{SQR}(X^2 + 1))/X)$
Inverse Hyperbolic Cotangent	$\text{ARCCOTH}(X) = \text{LOG}((X+1)/(X-1))/2$

## String Functions

The string functions manipulate string expressions.

<b>ASC</b>	Returns a numeric value that is the ASCII code of the first character of a string expression.
<b>CHR</b>	Returns the character that corresponds to a given ASCII code.
<b>HEX</b>	Returns a string expression that represents a hexadecimal value for a decimal argument.
<b>INSTR</b>	Searches for the first occurrence of a substring and returns the position where the match is found.
<b>LEFT</b>	Returns a string expression comprised of the requested, leftmost characters of a string expression.
<b>LEN</b>	Returns the number of characters in a string expression.
<b>MID</b>	Returns a substring from a given string expression.
<b>OCT</b>	Returns a string that represents the octal value of a decimal argument.
<b>RIGHT</b>	Returns a string expression comprised of the requested, rightmost characters in a string expression.

## Chapter 6:

### QW BASIC Statements, Commands, Functions, and Variables

6-1	Introduction
6-2	Chapter Format
6-3	ABS Function
6-3	ASC Function
6-4	ATN Function
6-5	AUTO Command
6-7	BEEP Statement
6-8	BLOAD Command/Statement
6-11	BSAVE Command/Statement
6-13	CALL Statement
6-16	CALLS Statement
6-17	CDBL Function
6-18	CHAIN Statement
6-23	CHDIR Statement
6-26	CHR\$ Function
6-26	CINT Function
6-27	CIRCLE Statement
6-31	CLEAR Statement
6-33	CLOSE Statement
6-34	CLS Statement
6-36	COLOR Statement (text)
6-39	COLOR Statement (graphics)
6-42	COM(n) Statement
6-43	COMMON Statement
6-46	CONT Command
6-47	COS Function
6-48	CSNG Function

**SPACE\$** Returns a string of spaces the length of a numeric expression.

**STR\$** Returns a string representation of a numeric expression.

**STRING\$** Returns a given length string whose characters all have the same ASCII code.

**VAL** Returns the numeric value of a string expression.

Vectra BASIC provides the following special functions:

### Special Functions

**ENVIRON\$** Displays the specified string from Vectra BASIC's environment table. (This table is used by the SHELL command.)

**ERR and ERL** Returns the error number and line number for the last-encountered program error.

**FRE** Returns the amount of free space after forcing a "garbage collection".

**PEEK** Returns the byte (decimal integer in the range 0 (eight zeros) to 255 (eight ones)) read from a memory location.

**PLAY** Returns the number of notes remaining in the Music Background buffer.

**USR** Calls an assembly-language subroutine.

**VARPTR** Returns the address of the first byte of data identified by a variable's name.

**VARPTR\$** Returns the character form for the memory address of a variable.

<b>6-49</b>	CSRLIN Function
<b>6-50</b>	CVI, CVS, CVD Functions
<b>6-51</b>	DATA Statement
<b>6-53</b>	DAT\$ Function
<b>6-54</b>	DAT\$ Statement
<b>6-56</b>	DEF FN Statement
<b>6-58</b>	DEF SEG Statement
<b>6-60</b>	DEF USR Statement
<b>6-62</b>	DEFINT/SG/DBL/STR Statements
<b>6-64</b>	DELETE Command
<b>6-66</b>	DIM Statement
<b>6-68</b>	DRAW Statement
<b>6-73</b>	EDIT Command
<b>6-74</b>	END Statement
<b>6-75</b>	ENVIRON\$ Statement
<b>6-77</b>	ENVIRON\$ Function
<b>6-79</b>	EOF Function
<b>6-81</b>	EOF Function (for Com Files)
<b>6-82</b>	ERASE Statement
<b>6-83</b>	ERDEV and ERDEV\$ Variables
<b>6-85</b>	ERR and ERL Variables
<b>6-88</b>	ERROR Statement
<b>6-89</b>	EXP Function
<b>6-90</b>	FIELD Statement
<b>6-93</b>	FILES Command/Statement
<b>6-95</b>	FIX Function
<b>6-96</b>	FOR...NEXT Statement
<b>6-100</b>	FRE Function
<b>6-101</b>	GET Statement
<b>6-102</b>	GET Statement (graphics)
<b>6-104</b>	GET and PUT Statements (for Com Files)
<b>6-105</b>	GOSUB...RETURN Statement
<b>6-106</b>	GOTO Statement
<b>6-109</b>	HEX\$ Function
<b>6-110</b>	IF Statement
<b>6-114</b>	INKEY\$ Function
<b>6-117</b>	INP Function
<b>6-118</b>	INPUT Statement
<b>6-122</b>	INPUT# Statement
<b>6-124</b>	INPUT\$ Function
<b>6-126</b>	INSTR Function

<b>6-127</b>	INT Function
<b>6-128</b>	IOCTL Statement
<b>6-129</b>	IOCTL\$ Function
<b>6-130</b>	KEY Statement
<b>6-134</b>	KEY(n) Statement
<b>6-136</b>	KILL Command/Statement
<b>6-138</b>	LEFT\$ Function
<b>6-139</b>	LEN Function
<b>6-140</b>	LET Statement
<b>6-141</b>	LINE Statement
<b>6-145</b>	LINE INPUT Statement
<b>6-146</b>	LINE INPUT# Statement
<b>6-148</b>	LIST and LLIST Command
<b>6-151</b>	LOAD Command
<b>6-153</b>	LOC Function
<b>6-154</b>	LOC Function (for COM files)
<b>6-155</b>	LOCATE Statement
<b>6-158</b>	LOF Function
<b>6-159</b>	LOF Function (for COM files)
<b>6-160</b>	LOG Function
<b>6-161</b>	LPOS Function
<b>6-162</b>	LPRINT and LPRINT USING Statements
<b>6-164</b>	LSET and RSET Statements
<b>6-166</b>	MERGE Command
<b>6-167</b>	MID\$ Function
<b>6-168</b>	MID\$ Statement
<b>6-170</b>	MKDIR Statement
<b>6-171</b>	MKIS MK\$ MKD\$ Functions
<b>6-173</b>	NAME Statement
<b>6-174</b>	NEW Command
<b>6-175</b>	OCT\$ Function
<b>6-177</b>	ON COM Statement
<b>6-179</b>	ON ERROR GOTO Statement
<b>6-180</b>	ON...GOSUB Statement
<b>6-181</b>	ON...GOTO Statement
<b>6-184</b>	ON KEY Statement
<b>6-186</b>	ON PEN Statement
<b>6-188</b>	ON PLAY Statement
<b>6-190</b>	ON STRIG Statement
<b>6-192</b>	ON TIMER Statement
	OPEN Statement

**6-197** OPEN "COM Statement  
**6-201** OPTION BASE Statement  
**6-202** OUT Statement  
**6-203** PAINT Statement  
**6-210** PEK Function  
**6-211** PEN Statement  
**6-212** PEN(n) Function  
**6-214** PLAY Statement  
**6-218** PLAY(n) Function  
**6-219** PMAP Function  
**6-220** POINT Function  
**6-222** POKE Statement  
**6-223** POS Function  
**6-224** PRESET Statement  
**6-226** PRINT Statement  
**6-229** PRINT USING Statement  
**6-235** PRINT# and PRINT# USING Statements  
**6-238** PSET Statement  
**6-240** PUT Statement  
**6-241** PUT Statement (graphics)  
**6-244** RANDOMIZE Statement  
**6-246** READ Statement  
**6-248** REM Statement  
**6-250** RENUM Command  
**6-252** RESET Command/Statement  
**6-253** RESTORE Statement  
**6-254** RESUME Statement  
**6-255** RETURN Statement  
**6-256** RIGHT\$ Function  
**6-257** RMDIR Statement  
**6-259** RND Function  
**6-260** RUN Command/Statement  
**6-262** SAVE Command  
**6-264** SCREEN Function  
**6-267** SCREEN Statement  
**6-272** SGN Function  
**6-273** SHELL Statement  
**6-276** SIN Function  
**6-277** SOUND Statement  
**6-281** SPACES Function  
**6-282** SPC Function

**6-283** SQR Function  
**6-284** STICK Function  
**6-285** STOP Statement  
**6-287** STR\$ Function  
**6-288** STRIG Statement  
**6-289** STRIG(n) Function  
**6-291** STRIG(n) Statement  
**6-293** STRING\$ Function  
**6-294** SWAP Statement  
**6-295** SYSTEM Command/Statement  
**6-296** TAB Function  
**6-297** TAN Function  
**6-298** TIMES Function  
**6-299** TIMES Statement  
**6-300** TIMER Function  
**6-301** TIMER Statement  
**6-303** TRON/TROFF Statements  
**6-305** USR Function  
**6-306** VAL Function  
**6-307** VARPTR Function  
**6-310** VARPTR\$ Function  
**6-312** VIEW Statement  
**6-316** VIEW PRINT Statement  
**6-317** WAIT Statement  
**6-318** WHILE...WEND Statement  
**6-320** WIDTH Statement  
**6-322** WINDOW Statement  
**6-326** WRITE Statement  
**6-327** WRITE# Statement



# **6**

## **Vetra BASIC Statements, Commands, Functions, and Variables**

### **Introduction**

This chapter contains a comprehensive listing of the commands, statements, functions, and variables that Vectra BASIC provides.

The distinction between commands and statements is mainly traditional. In general, commands operate on programs, and you usually enter them in Direct Mode. Statements direct the flow of control within a BASIC program.

Functions are predefined operations that perform a specific task. They return a numeric or string value. You can put the built-in functions and variables to immediate use.

## Chapter Format

The statement and command descriptions take the following form:

- Format:** Shows the correct syntax for that instruction.
- Purpose:** Describes the instruction and what it does.
- Remarks:** Provides details on the instruction's use and supplies pertinent notes and comments.
- Example:** Gives an example of the instruction's use.
- Since most of the functions perform familiar operations (such as taking the square root of a number or returning the sine of an angle), their description is brief.
- Format:** Shows the correct syntax for the function.
- Action:** Describes what the function does.
- Example:** Gives sample program segments that demonstrate the function's use.

Appendix B provides syntax diagrams for all the Vectra BASIC instructions.

## ABS Function

- Format:** ABS (X)
- Action:** Returns the absolute value of the expression X.
- Example:**
- ```
PRINT ABS(-5 * 7)
35
OK
```

## ASC Function

- Format:** ASC(X\$)
- Action:** Returns a numeric value that is the ASCII code of the first character in the string X\$. (Appendix B lists the ASCII codes.)
- If X\$ is the null string, an **Illegal function call** occurs.
- See the **CHR\$** function for ASCII-to-string conversions.
- Example:**
- ```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
84
OK
```

## ATN Function

**Format:** ATN(x)

**Action:** Returns the arctangent of x. The result is in radians and ranges between -PI/2 and PI/2.

**Note** To convert radians to degrees, multiply by 180/PI, where PI = 3.141593.

The expression x may be any numeric type. Vectra BASIC evaluates ATN in single-precision arithmetic.

To achieve a double-precision result, x must be defined as a double-precision variable and Vectra BASIC must be invoked with the /D switch, or the results of the function must be stored in a double-precision variable, as in `J% = ATN(x)`.

### Example:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
1.249046
OK
```

## AUTO Command

**Format:** AUTO (line#, increment)

**Purpose:** Generates a line number automatically when you press the [Enter] key. You normally use this command when you are entering a program to free yourself from typing each line number.

**Remarks:** AUTO begins numbering at line# and increments each subsequent line number by increment. The default setting for both values is 10. If you follow line# with a comma but omit the increment, Vectra BASIC uses the increment specified in the last AUTO command.

When the AUTO command generates a line number that is already being used, Vectra BASIC prints an asterisk after the number to warn you that any characters you type will replace the existing line. If this is not your intent, you may press the [Enter] key to preserve the old line and generate the next line number.

### Note

Pressing the [Enter] key must be your first action after the warning asterisk appears. If you press a character before pressing the [Enter] key, Vectra BASIC replaces the current line with that character.

Simultaneously pressing [CTRL] [Break] stops the automatic generation of line numbers. Since pressing the [Enter] key to end a line generates a new number for the next line, Vectra BASIC discards the line in which you press [CTRL] [Break]. However, when the line in which you type [CTRL] [Break] has an asterisk after the line number (showing that the line currently exists), Vectra BASIC preserves the old line. Vectra BASIC returns control to the command level.

**Examples:**

This first example generates line numbers beginning at 10 and incrementing by 10. (10 is the default value for both the starting line number and the increment.):

`AUTO`

The next example generates the line numbers 100, 150, 200, etc.:

`AUTO 100, 50`

The last example generates line numbers beginning with 1000 and increasing by 50 at each step. (The example assumes that this command follows the preceding command where the increment was 50.):

`AUTO 1000,`

**Note**

The BASIC compiler offers no support for this command.

**BEEP Statement**

**Format:** `BEEP`

**Purpose:** Sounds the computer's bell.

**Remarks:** This function is equivalent to `PRINT CHR$(7)`, where 7 is the decimal value of the ASCII bell character.

**Example:** `20 IF USER.ERROR = 245 THEN BEEP`

## BLOAD Command/Statement

**Format:**

BLOAD *filename* [*,offset*]

**Purpose:**

Loads the specified memory image file from disc into your computer's memory.

**Remarks:**

*filename* is a string expression that contains the name of the file. It may contain an optional drive designator and path.

If no drive designator is included in *filename*, Vectra BASIC uses the current drive. If no path is specified, Vectra BASIC searches the current directory for *filename*.

Vectra BASIC will supply the filename extension `.BAS` if no extension is specified.

If *filename* is a literal, you must enclose the name in quotation marks.

*offset* is a numeric expression that returns an unsigned integer which may range between 0 and 65535. This is used in conjunction with a DEF SEG statement to specify an alternate location where loading begins.

As a command, you can use BLOAD to load assembly-language routines immediately into memory. A program can use BLOAD as a statement to selectively load assembly-language routines.

The BLOAD statement loads a program or data file (which you saved as a memory image file) anywhere in memory. A **memory image file** is a byte-for-byte copy of what was originally in memory. See the BSAVE command in this chapter for information about saving memory files.

When you omit the *offset* parameter, Vectra BASIC uses the segment address and offset that are contained in the file. (That is, the address you specified in the BSAVE statement when you created the file.) Therefore, Vectra BASIC loads the file back to the same location from which it was originally saved.

When you give an offset, Vectra BASIC uses the segment address from the most recently executed DEF SEG statement. Therefore, a program should execute a DEF SEG statement before it executes a BLOAD statement. If Vectra BASIC fails to encounter a DEF SEG statement, it uses the BASIC Data Segment (DS) as the default address.

**Caution**

Since BLOAD never performs an address range check, you may load a file anywhere in memory. You must be careful, therefore, to avoid loading a file over the Vectra BASIC interpreter program or the MS-DOS operating system.

**Example:**

The following example sets the segment address at 6000 Hex and loads PROG1 at F000:

```
10 REM Load subroutine at 6F000
20 DEF SEG = &H6000 'Set segment
   to 6000 Hex
30 BLOAD "PROG1", &HF000 'Load PROG1
```

These program lines load a graphics image onto a graphics screen. Be sure that you use the same graphics mode that you used when you created and saved the picture.

```
3000 REM Load picture
3010 DEF SEG = &HB800
3020 BLOAD "DRAWING.PIC"
```

**Note**

Pictures saved with graphics software packages sometimes include a few bytes of extra information at the beginning of the file. If pictures don't seem to load properly, try using a smaller value in DEF SEG, and adding an *offset* parameter to the BLOAD statement.

For example, if a medium resolution graphic appears to be offset by 8 bytes (32 pixels), you would use:

```
2000 DEF SEG = &HB7FF
2010 BLOAD "DRAWING.PIC", 8
```

**Note**

The BASIC compiler offers no support for this command.

**BSAVE Command/Statement**

**Format:**

BSAVE *filename*, *offset*, *length*

**Purpose:**

Saves the contents of the specified area of memory as a disc file. (Also see the BLOAD statement.)

**Remarks:**

*filename* is a string expression that contains the name of the file. It may contain an optional drive designator and path.

If no drive designator is included in *filename*, Vectra BASIC uses the current drive. If no path is specified, Vectra BASIC searches the current directory for *filename*.

Vectra BASIC will supply the filename extension .BAS if no extension is specified.

If *filename* is a literal, you must enclose the name in quotation marks.

*offset* is a numeric expression that returns an unsigned integer which may range between 0 and 65535. This is the offset address into the segment that you declared in the last DEF SEG statement. It specifies the exact location of the first byte of memory that is saved to disc.

*length* is a numeric expression that returns an unsigned integer which may range between 1 and 65535. This gives the length in bytes of the memory image file that you want to save.

The syntax for BSAVE requires all three parameters: *filename*, *offset*, and *length*. If you enter an improper *filename*, a Bad file name error occurs. Omitting *offset* or *length* produces a Syntax error. Under any of these circumstances, Vectra BASIC cancels the BSAVE operation.

Since the address given in the most recently executed DEF SEG statement determines the starting point from which Vectra BASIC calculates the offset, you should execute a DEF SEG statement before you execute a BSAVE statement.

Example:

The following example saves 256 bytes, beginning at 0F000, in file MEMIMAGE:

```
10 REM SAVE MEMIMAGE
20 DEF SEG = &H6000
30 BSAVE "MEMIMAGE", &HF000, 256
```

These program lines save other graphics screen as a file. The graphic can be redisplayed using the BLOAD statement.

```
2000 REM Save Picture
2010 DEF SEG = &HB800
2020 BSAVE "ARTWORK.PIC", 0, 16000
```

Note

The BASIC compiler offers no support for this command.

CALL Statement (for Assembly Language Subroutines)

Format:	CALL <i>varname</i> [( <i>Argument</i> [, <i>Argument</i> [, . . .]])]
Purpose:	Calls an assembly-language subroutine.
Remarks:	<p><i>varname</i> contains the segment offset that is the starting point in memory of the called subroutine. It cannot be an array variable name. You must assign the segment offset to the variable before you use the CALL statement.</p> <p><i>argument</i> is a variable or constant that is being passed to the subroutine. No literals are allowed. You must separate the items in the list with commas.</p>

The CALL statement is the recommended way of calling machine-language programs with Vectra BASIC. You should avoid the USR function. See Appendix C, Assembly Language Subroutines.

The CALL statement generates the same calling sequence that is used by Microsoft® FORTRAN and Microsoft® BASIC compilers.

When the CALL statement executes, Vectra BASIC transfers control to the routine via the segment address given in the last DEF SEG statement and the segment offset specified by the *varname* parameter of the CALL statement. You may return values to the calling program by including within the list of arguments variable names to receive the results.

Example:

The following program loads an assembly-language subroutine into memory, then calls it. The DATA statements contain the assembled code with byte pairs inverted. (This is an easy way of loading the code into memory.) The call to VARPTR locates the starting location of the first byte of the code, then the subroutine is called using that offset. It is important to include the VARPTR call just prior to the subroutine call since you need the current location and the array containing the code may move around in memory as Vectra BASIC defines more variables.

```
10 DATA 4H8B55, 4H8BEC, 4H065E, 4HD88C
20 DATA 4H0789, 4HCASD, 4H0002
30 DIM GETDS%(6): FOR INDEX=0 TO 6
40 READ GETDS%(INDEX): NEXT INDEX
50 ADDR1=VARPTR(GETDS%(0)): CALL ADDR1(ADDR1)
60 PRINT HEX$(ADDR1)
```

A copy of the assembly-language subroutine follows:

0000	code	segment byte public 'code'
		public gets
		assume cs code
		; subroutine: return DS to calling program
0000	gets	proc far
0000 55		push bp
		; Save current bp register
0001 8B EC		mov bp,sp
		; Use bp as stack pointer
0003 8B 5E 06		mov bx,[bp+6]
		; Load addr of variable into bx
0006 8C D8		mov ax,ds
		; Load value of DS into ax
0008 89 07		mov [bx],ax
		; Store value of DS (in ax) into variable
000A 5D		pop bp
		; Recall original value of bp
000B CA 0002		ret 2
		; Return to main prog. with ; cleanup
000E	gets	endp
000E	code	ends
		end

Note

Refer to the BASIC compiler manual for differences between the interpretive and compiled versions of BASIC when using the CALL statement.

## CALLS Statement

**Format:**      `CALLS variable (Argument list)`

**Purpose:**      Calls a subroutine with segmented addresses.

**Remarks:**    The CALLS statement resembles the CALL statement, except the segmented addresses of all arguments are passed. A CALL statement passes unsegmented addresses.

As with the CALL statement, CALLS uses the segment address defined by the most recently executed DEF SEG statement to

- locate the routine being called.

### Note

For more information, refer to Appendix C, "Assembly Language Subroutines".

## CDBL Function

**Format:**      `CDBL(x)`

**Action:**      Converts *x* to a double-precision number.

**Example:**

```
10 A = 454.67
20 PRINT A; CDBL(A)
RUN
454.67 454.6700134277344
OK
```

## CHAIN Statement

### Format:

CHAIN [MERGE] *filename* [, *line*] [, ALL] [, DELETE] [*range*]

### Purpose:

Calls a program and passes variables to it from the current program.

### Remarks:

*filename* is a string expression that contains the name of the file. It may contain an optional drive designator and path.

If no drive designator is included in *filename*, Vectra BASIC uses the current drive. If no path is specified, Vectra BASIC searches the current directory for *filename*.

Vectra BASIC will supply the filename extension .BAS if no extension is specified.

If *filename* is a literal, you must enclose the name in quotation marks.

In the example:

```
CHAIN "PROG1"
```

Vectra BASIC searches the currently active disc directory for the file PROG1.BAS. When it locates the file, it loads then executes the program. Once the program resides in memory, you may list and modify it.

If Vectra BASIC fails to locate the file, it prints a **File not found** error message, and when no **ON ERROR** statement is active, halts execution and returns the user to the command line.

*line* is either a line number or an expression which evaluates to a line number, in the called ("chained-to") program. It becomes the starting point for executing the called program. When you omit this parameter, Vectra BASIC begins executing the called program at the first line. The following statement begins executing PROG1 at line 1000:

```
CHAIN "PROG1", 1000
```

If Vectra BASIC fails to find the given line number, an **Undefined line number** error results.

Since *line* refers to a line in another program, a **RENUM** command has no effect on it. (**RENUM** only affects line numbers in the current (or calling) program.)

During the chaining process, any files that were previously opened remain open.

The **ALL** option passes every variable in the current program to the called program. When you omit this parameter, the current program must contain a **COMMON** statement to list the variables that are being passed. An example of a **CHAIN** statement with the **ALL** option is:

```
CHAIN "PROG1", 1000, ALL
```

The arguments for the **CHAIN** statement are position dependent. For example, when you use the **ALL** option but omit the starting line, you must include a comma to hold the place for the line parameter. That is, **CHAIN "NEXTPRG", , ALL** is correct while **CHAIN "NEXTPRG", ALL** is illegal. (In the latter statement, Vectra BASIC assumes **ALL** is a variable name for a line number expression.)

Including the **MERGE** option allows a subroutine to be brought into the Vectra BASIC program as an overlay. That is, Vectra BASIC merges the called program with the current program. The called program must be in ASCII format before you can merge it.

```
CHAIN MERGE "OVERLAY", 1000
```

When using the **MERGE** option, you should place any user-defined functions before any **CHAIN MERGE** statements in that program. If they are not defined prior to the merge, they remain undefined after the merge operation is completed.

The **CHAIN** statement with **MERGE** option leaves files open and preserves the current **OPTION BASE** setting.

When you omit the **MERGE** option, the **CHAIN** statement does not preserve variable types or user-defined functions for use by the called program. That is, you must reissue any **DEFINT**, **DEFSTR**, **DEFDBL**, **DEFSTR**, or **DEF FN** statements within the called program.

After an overlay is brought in and finishes processing, you may delete it with the **DELETE** option in a new **CHAIN** statement.

- This allows Vectra BASIC to bring in a new overlay if one is needed.

```
CHAIN MERGE "OVRLAY2", 1000, DELETE 1000-5000
```

The above statement deletes lines 1000 to 5000 in the current program, merges in the file **OVRLAY2.BAS**, and resumes execution at line number 1000.

When your program contains a **CHAIN** statement that uses the **DELETE range** option, you should use the **RENUM** command with caution. (The **RENUM** command affects the line numbers in *range* since they refer to lines in the current program.)

### Note

The **CHAIN** statement does a **RESTORE** before running the chained program. Therefore, the next **READ** statement accesses the first item in the first **DATA** statement that the program contains. The read operation does not continue from where it left off in the chaining program.

### Examples:

The first example demonstrates the **CHAIN** statement in its simplest form.

```
5 REM -----THIS IS PROGRAM 1 -----
10 REM THIS EXAMPLE PASSES VARIABLES
15 REM USING THE "COMMON" STATEMENT
20 REM SAVE THIS MODULE ON DISK AS "PROG1" USING
  THE A OPTION
30 DIM A$(2), B$(2)
40 COMMON A$(2), B$(2)
50 A$(1) = "VARIABLES IN COMMON MUST BE ASSIGNED"
60 A$(2) = "VALUES BEFORE CHAINING,"
70 B$(1) = " " : B$(2) = " "
80 CHAIN "PROG2"
90 PRINT B$(1) : PRINT B$(2)
100 END
```

```
5 REM -----THIS IS PROGRAM 2 -----
10 REM STATEMENT 30 ABOVE "DIM A$(2), B$(2)"
  MAY ONLY BE EXECUTED ONCE.
20 REM HERE, IT DOES NOT APPEAR IN THIS MODULE.
30 REM SAVE THIS MODULE ON THE DISK AS "PROG2"
  USING THE A OPTION.
40 COMMON A$(2), B$(2)
50 PRINT A$(1) : PRINT A$(2)
60 B$(1) = "NOTE HOW THE OPTION OF SPECIFYING A
  STARTING LINE"
70 B$(2) = "WHEN CHAINING AVOIDS THE DIM
  STATEMENT IN 'PROG1'."
80 CHAIN "PROG1", 90
90 END
RUN "PROG1" Enter
VARIABLES IN COMMON MUST BE ASSIGNED
VALUES BEFORE CHAINING.
NOTE HOW THE OPTION OF SPECIFYING A
STARTING LINE
WHEN CHAINING AVOIDS THE DIM
STATEMENT IN 'PROG1'.
```

The next example demonstrates some of the options that you may use with the CHAIN statement.

```

5 REM -----MAINPRG -----
10 REM THIS EXAMPLE USES THE MERGE, ALL, AND
  DELETE OPTIONS.
20 REM SAVE THIS MODULE ON THE DISC AS "MAINPRG".
30 A$ = "MAINPRG"
40 CHAIN MERGE "OVRLAY1", 1010, ALL
50 END

1000 REM SAVE THIS MODULE ON DISC AS "OVRLAY1"
  USING THE A OPTION.
1010 PRINT A$; " HAS CHAINED TO OVRLAY1."
1020 A$ = "OVRLAY1"
1030 B$ = "OVRLAY2"
1040 CHAIN MERGE "OVRLAY2", 1010, ALL,
  DELETE 1000-1050
1050 END

1000 REM SAVE THIS MODULE ON DISC AS "OVRLAY2"
  USING THE A OPTION.
1010 PRINT A$; " HAS CHAINED TO "; B$; " ."
RUN "MAINPRG" 
MAINPRG HAS CHAINED TO OVRLAY1.
OVRLAY1 HAS CHAINED TO OVRLAY2.
OK

```

### Note

The BASIC compiler offers no support for the ALL, MERGE, and DELETE options to the CHAIN statement. If you want to maintain compatibility with the BASIC compiler, you should pass variables with the COMMON statement and avoid overlays.

## CHDIR Statement

**Format:** CHDIR *path*

**Purpose:** Changes the current directory.

**Remarks:** *path* is a string expression (not exceeding 63 characters) that identifies the new directory.

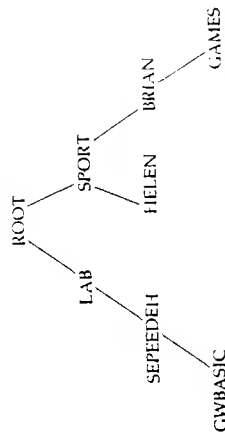
Vectra BASIC permits tree-structured directories as allowed by MS-DOS version 2.0 and later versions. The CHDIR statement changes the current directory to another directory within the tree-structured hierarchy.

### Examples:

This example selects INVENTORY to be the current, default directory on drive A:

```
CHDIR "A:INVENTORY"
```

The following examples refer to this tree-structured directory:



You may change from any subdirectory to the ROOT directory with this statement:

```
CHDIR "\"
```

To change from the `ROOT` directory to `SEPEDEH` requires this statement:

```
CHDIR "LAB\SEPEDEH"
```

To change from the directory `SPORT` to `HELEN` requires this statement:

```
CHDIR "HELEN"
```

The notation `..` refers to the directory that resides immediately above the current directory. Therefore, you may change from the directory `BRIAN` to `SPORT` with this statement:

```
CHDIR ".."
```

To change from the directory `BRIAN` to `HELEN`, use the statement:

```
CHDIR "... \HELEN"
```

## CHRS Function

### Format:

`CHR$(i)`

### Action:

Returns the character that corresponds to a given ASCII code.

`i` may range from 0 to 255. The text screen character set contains all 256 characters. The graphics screen character sets contain only characters 0–127.

You normally use `CHR$` to send special characters to the computer, a file, or a device. For example, you could send the BELL character (`CHR$(7)`) as a preface to an error message.

See the `ASC` function for ASCII-to-numeric conversions.

See Appendix B for a list of characters, and for further information on character sets.

### Examples:

```
PRINT CHR$(66)  
B  
Dk
```

## CINT Function

**Format:** CINT(*x*)

**Action:** Converts *x* to an integer by rounding off the fractional part. *x* must be within the range of -32768 to 32767. If *x* is outside this range, an **Overflow** error occurs.

See the **DBL** and **DSNG** functions for converting numbers to double-precision and single-precision data types. See also the **FIX** and **INT** functions, both of which return integers.

**Example:**

```
PRINT CINT(45.67)
46
OK
```

## CIRCLE Statement

**Format:** CIRCLE [STEP] (*x*, *y*), *r* [, *color* [, *start*, *end* [, *aspct*]]]

**Purpose:** Draws an ellipse on the screen with center *x,y* and radius *r*.

**Remarks:** *x,y* is the coordinate pair for the center of the ellipse. The center of the ellipse becomes the "last point referenced" after the ellipse is drawn.

You may give the *x* and *y* coordinates as absolute numbers, or you may use the **STEP** option to give the coordinates in relative form. In this case, the *x,y* parameter takes the following form:

STEP (*xoff*/*set*, *yoff*/*set*)

For example, if the last-referenced point were (15,5),  
STEP (10, 5) references the point at (25,10).

*r* is the radius of the ellipse in pixels.

This command is only valid in the medium and high resolution graphics modes. The medium resolution screen measures 320 pixels (or dots) by 200 pixels. The high resolution screen measures 640 pixels by 200 pixels.

*x*, *y* and *r* may range from -32768 to 32767. Values are rounded to integers before the circle is drawn.

Circles may be drawn larger than the screen boundary; they will be clipped at the screen edges. If *x* or *y*, or both, is outside the screen boundary, the **CIRCLE** statement draws the circle centered on the off-screen point. The resulting figure may be partially or totally off the screen.

For example, CIRCLE (160, 50), 75 produces an arc at the top of the screen.

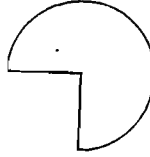
*color* is optional. In medium resolution, it references a color of the palette selected by the **COLOR** statement. 0 produces the background color; 1-3 reference the three available colors. The default value is 3. Any value between 3 and 255 will produce color 3; any value greater than 255 will produce an **illegal function call** error message.

In high resolution, the color of 0 or 2 produces the background color; any other value between 1 and 255 produces the foreground color set by the **COLOR** statement. A value greater than 255 produces an **illegal function call** error.

*start* and *end* are angles in radians; they may range from  $-2 * \pi$  to  $2 * \pi$ . (Using  $\pi$  requires the statement **PI = 3.141593**.) These angles specify where drawing the ellipse begins and ends. The start angle may be less than the end angle. If either angle is negative, a line connects that end of the ellipse to the center point. For example,

```
10 PI = 3.141593
20 CIRCLE (150, 60), 50, -PI, -PI/2
```

draws the following figure:



The angles are positioned with 0 at the right, and increase counterclockwise. This is standard mathematical practice.

## Note

To convert degrees to radians, multiply degrees by .0174532.

*aspect* is the aspect ratio; that is, the ratio of the *x* radius to the *y* radius. For a circle to appear round on the screen, the default value for *aspect* is 5/6 in medium resolution and 5/12 in high resolution.

When *aspect* is less than or equal to the default value, the *x* radius is equal to *r*, and the *y* radius equals *aspect* \* *r*. When *aspect* is greater than the default value, the *y* radius equals *r* and the *x* radius equals *aspect*/*r*.

The following program draws four ellipses. The blue ellipse appears as a circle on the screen. It uses the default value for *aspect*. The magenta ellipse uses an *aspect* of 1; its *x* radius and *y* radius both equal 50 pixels. The first white ellipse uses an *aspect* of 2; its *y* radius equals 50 pixels and its *x* radius equals 25 pixels (50/2). The final ellipse uses an *aspect* of .2; its *x* radius equals 50 pixels, and its *y* radius equals 10 pixels (50 \* .2). The **LINE** statements provide reference marks every 10 pixels.

```
10 SCREEN 1:COLOR 0,1:CLS
20 CIRCLE (160,100),50,1:REM BLUE
30 CIRCLE (160,100),50,2,1:REM MAGENTA
40 CIRCLE (160,100),50,3,1:REM WHITE
50 CIRCLE (160,100),50,3,1:REM WHITE
60 FOR Y = 50 TO 100 STEP 10
70 LINE (150,Y)-(162,Y)
80 NEXT Y
90 FOR X=160 TO 210 STEP 10
100 LINE (X,98)-(X,102)
110 NEXT X
```

### Example:

If the last-referenced point were (10,20), then both of the following statements draw a circle at (100,50) with a radius of 50 pixels.

```
770 CIRCLE STEP (90,30),50
770 CIRCLE (100,50),50
```

```
10 REM Draw a bunch of circles, in three colors
20 SCREEN 1
30 COLOR 0,0
40 KEY OFF
50 CLS
60 FOR I=1 TO 100 STEP 4
70   CIRCLE (160+I,1+50),20+I,1
80   CIRCLE (160-I,1+50),20+I,2
90   CIRCLE (160,1+50),20+I
100 NEXT I
```

## CLEAR Statement

**Format:** CLEAR (, [expression1] [, expression2])

**Purpose:** Sets all numeric variables to zero and all string variables to the null string, closes all files, and, optionally, sets the end of memory and the amount of stack space.

**Remarks:** *expression1* sets the maximum number of bytes for the Vectra BASIC workspace. When you omit this parameter, Vectra BASIC uses all available memory up to 64K.

*expression2* sets aside stack space for Vectra BASIC. When you omit this parameter, Vectra BASIC sets aside either 512 bytes or one-eighth of the available memory, whichever is smaller.

The CLEAR statement performs the following functions:

- Frees all memory used for data without erasing the program currently in memory
- Closes all files
- Clears all COMMON and user variables
- Resets the stack and string space
- Releases all disc buffers
- Resets the dimensions for all arrays to the default setting of 10
- Resets all numeric variables and arrays to zero
- Resets all string variables and arrays to null
- Clears definitions set by any DEF statements. (This includes DEF FN, DEF SEG, and DEF USR, as well as DEF INT, DEF SNO, DEF DBL, and DEF STR.)
- Stops any SOUND or PLAY statements that are playing, and resets PLAY to Music Foreground.
- Sets PEN and STRIG to OFF.
- Sets DRAW scale and colors to their default values.

Also see the ERASE statement, which deletes arrays. It can reclaim some needed storage space without erasing all the information in the workspace.

### Examples:

The first example clears all data from memory without erasing the program:

```
CLEAR
```

The next statement clears all data and sets the maximum workspace size to 32K bytes:

```
CLEAR, 32768
```

The next example clears all data and sets the size of the stack to 2000 bytes:

```
CLEAR, 2000
```

The last example clears all data and sets the maximum workspace size to 32K bytes and the stack size to 2000 bytes:

```
CLEAR, 32768, 2000
```

### Note

If you intend to compile your program, consult the BASIC compiler manual for differences in implementation between the compiled and interpretive version of this command.

## CLOSE Statement

### Format:

```
CLOSE [(#) filename [, (#) filename, ...]]
```

### Purpose:

Concludes input or output to a disc file or device.

### Remarks:

*filename* is the number you gave the file when you opened it.

A CLOSE statement with no arguments closes all open files and devices.

The association between a particular file and its file number ceases when the file is closed. Therefore, you may then reopen the file using the same or a different file number. Similarly, you may use the freed file number to open a new file.

A CLOSE for a sequential output file writes the final buffer of output to the file.

The following instructions close all disc files automatically:

- END
- NEW
- RESET
- CLEAR
- RUN without the R option
- SYSTEM

The STOP statement, however, never closes any disc files.

### Example:

```
100 OPEN "0", #2, "OUTFILE"
110 PRINT #2, CNAME$, ADDRESS$, ZIP$, PHONE$
120 CLOSE #2
```

## CLS Statement

**Format:**

CLS

**Purpose:**

Clears the screen of the current screen window.

**Remarks:**

After the screen or viewport is cleared, the cursor is returned to the "home" position for the screen or viewport. For the text cursor, the home position is the upper left corner of the screen or viewport.

In graphics mode, the "last referenced point" is reset to the center of the screen or viewport. In medium resolution this position is (160,100); in high resolution it is (320,100) with no viewports. This point becomes the "last referenced point" for future graphics commands.

If the computer is in text mode (SCREEN 0), this statement clears the active page, which may not be the page which is displayed on the screen. See the SCREEN statement for more information.

With a color monitor adapter, CLS clears the screen to the background color set by a COLOR statement.

If a VIEW statement is in effect, only the active window is cleared.

Other commands can clear the screen:

**CTRL** **L**  
**CTRL** **Home**

SCREEN statements which change from one screen mode to another

WIDTH statements which cause a change from one screen mode to another

**Example:**

This statement clears the entire screen:

```
10 CLS
```

## COLOR Statement (Text mode)

- Format:** COLOR (*foreground*) [, (*background*) [, (*border*) ] ]
- Purpose:** Sets the foreground, background, and border colors for the color monitor adapter, or changes the character modes and background for monochrome adapters.
- Remarks:** *foreground*, *background*, and *border* are numeric expressions. *foreground* can range from 0 to 31. It sets the foreground color or character mode. *background* can range from 0 to 7. It sets the background color. *border* can range from 0 to 15. It sets the border color. Any of these parameters may be omitted; an omitted parameter assumes its previous value.

### Note

When you change the border color, it changes immediately. Changing the background and foreground colors affects all subsequent screen output, but does not change characters already on the screen. To change the entire screen to a background color, follow the COLOR statement with a CLS statement.

The COLOR statement has different effects depending on your video display adapter. Many video display adapters (including the HP Multimode card) do not "overscan"; that is, they display only the actual screen portion, and the border area always remains black. The *border* parameter in a COLOR statement will take effect, but will not be visible on your monitor.

### Monochrome adapter

These values can be used for the foreground:

- 0 Black characters on light background. (Must be used with a light background color, that is, background 7.)
- 1 Underlined character, in light on black.
- 2-7 Light characters on dark background.

### Note

"Light" means whatever color your monochrome monitor produces: white, green or amber.

Characters can be printed in high intensity by adding 8 to the foreground color; to make characters blink, add 16 to the foreground color.

- COLOR 1,0 Underlined, light on black.
- COLOR 9,0 Underlined, high intensity, light on black.
- COLOR 17,0 Underlined, blinking, light on black.
- COLOR 25,0 Underlined, high intensity, blinking, light on black.

In monochrome, these are the values for the background:

- 0-6 Black
- 7 Light background color with black characters. The foreground (character) color must be black: 0, 8, 16 or 24.
- COLOR 0,7 Black letters on a light background.
- COLOR 16,7 Blinking black letters on a light background.

### Note

A COLOR statement that attempts to put black characters on a black background, or light characters on a light background is ignored. No error message is printed. COLOR statements that would produce mixed colors on a color adapter can run on a Vectra with a monochrome board and still produce readable text.

### Color adapter

The colors are:

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	Bright White

The foreground and border may be any of these colors; the background may be only colors 0-7. The colors 0-7 are darker shades of colors 8-15.

To make the foreground characters blink, add 16 to the color value. COLOR 4 produces red characters, COLOR 20 produces blinking red characters

### Example:

```
10 SCREEN 0
20 COLOR 5,1:CLS
30 LOCATE 1,5:PRINT "Date:";
40 COLOR 1,5:PRINT DATE$;
50 COLOR 3
60 PRINT "    Time:";
70 COLOR 3,1:PRINT TIME$
```

## COLOR Statement (Graphics modes)

**Format:** COLOR {color1} [,palette]

**Purpose:** Sets the background color and selects the foreground palette in medium resolution; selects the foreground color in high resolution.

**Remarks:** color specifies the color for the background in medium resolution, and the foreground color in high resolution. It can be an integer between 0 and 15. The colors are:

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	Bright white

Colors 0-7 are lower intensity; colors 8-15 are higher intensity. A low intensity background color results in low intensity foreground colors; a high intensity background results in high intensity values for its foreground palette. (The OUT statement provides a way to alter this effect.)

palette selects one of two palettes in medium resolution. It has no effect in high resolution.

Each palette consists of three colors that can be used for the graphics statements `CIRCLE`, `DRAW`, `LINE`, `PAYNT`, `PRESET` and `PSET`. For each of these statements you specify the color that will be used as a number between 0 and 3. The values 0-3 in these statements select the corresponding colors in this chart, depending on the palette chosen by the `COLOR` statement:

Color	palette 0	palette 1
0	background	background
1	green	cyan
2	red	magenta
3	brown	white

The default color for all graphics statements (except `PRESET`) is color 3.

If a parameter is omitted from a `COLOR` statement, the current background or palette is not changed.

In medium resolution, the text color can be changed by the statement:

```
POKE 4H4E,color
```

where *color* is 1-3. A *color* of 0 should never be used in this statement.

In graphics modes, the `COLOR` statement immediately changes the background color and palette to the chosen values.

`POKE 4H4E,color` affects subsequent text which is sent to the screen, but does not affect the color of text which is already on the screen.

## Examples:

This example uses the "Box Fill" parameter of the `LINE` statement to create colored boxes on the screen for each of the colors in palette 1. Then it changes the background colors in a `FOR...NEXT` loop.

```
10 SCREEN 1
20 COLOR 0,1
30 CLS
40 KEY OFF
50 LINE (10,10)-(100,180),1,BF
60 LINE (110,10)-(200,180),2,BF
70 LINE (210,10)-(300,180),,BF
80 LOCATE 12,6:PRINT "cyan";
90 LOCATE 12,17:PRINT "magenta";
100 LOCATE 12,30:PRINT "white"
110 LOCATE 23,1
120 FOR C = 0 TO 15
130 COLOR C
140 FOR J=1 TO 2000:NEXT J
150 NEXT C
160 COLOR 0
```

This example creates colored circles and fills them with the `PAYNT` statement. It uses `POKE 4H4E,color` to print colored text next to the circles.

```
10 SCREEN 1
20 COLOR 1,0
30 CLS
40 CIRCLE (50,50),40,1
50 PAYNT (50,50),1
60 POKE 4H4E,1:LOCATE 7,15:PRINT "green"
70 CIRCLE (50,100),40,2
80 PAYNT (50,100),2
90 POKE 4H4E,2:LOCATE 13,15:PRINT "red"
100 CIRCLE (50,150),40,3
110 PAYNT (50,150),3
120 POKE 4H4E,3:LOCATE 19,15:PRINT "brown"
130 LOCATE 23,1
```

## COM (n) Statement

**Format:** COM(n) ON  
COM(n) OFF  
COM(n) STOP

**Purpose:** Enables, disables, or suspends event trapping of communications activity on the specified channel.

**Remarks:** *n* is the number of the communications channel. The permissible values are 1 and 2.

- The COM(n) ON statement enables communications event trapping by an ON COM statement. While trapping is enabled, and if you have given a non-zero program line number in the ON COM statement, Vectra BASIC checks between every statement to see if any activity has occurred on the communications channel. When activity occurs, Vectra BASIC executes the ON COM statement.

### Note

See the ON COM statement for details on trapping communication events.

COM(n) OFF disables communications event trapping. If an event occurs, Vectra BASIC ignores it.

COM(n) STOP disables communications event trapping but if an event occurs, Vectra BASIC "remembers" it and executes an ON COM statement as soon as you again enable trapping.

### Example:

This example enables event trapping for communications activity on channel 1:

```
10 COM(1) ON
```

## COMMON Statement

**Format:** COMMON variable ( , variable ) ...

**Purpose:** Passes variables to a chained program.

**Remarks:** *variable* is the name of the passed variable. You specify array variables by appending a pair of parentheses "( )" to the variable's name.

The Vectra BASIC interpreter accepts the number of dimensions for an array as in:

```
COMMON EMPLOYEE(3)
```

but treats it as equivalent to:

```
COMMON EMPLOYEE()
```

Also, the number in parentheses is the number of dimensions, not the dimensions themselves. For example, EMPLOYEE(3) could correspond to either of the following DIM statements:

```
DIM EMPLOYEE(20,4,2)
```

or

```
DIM EMPLOYEE(10,5,12)
```

You use the COMMON statement in conjunction with the CHAIN statement. You pass variables in the main program to variables in the chained program by listing each variable name in a COMMON statement.

Although **COMMON** statements may appear anywhere within a program, good programming practice dictates grouping them at the program's beginning.

You cannot name the same variable in multiple **COMMON** statements.

When you want to pass all the variables within a program, you should use the **CHAIN** statement with the **ALL** option and omit the **COMMON** statement.

**Example:**

```
10 COMMON CUST$,A,F,C)
20 PRINT CUST$,A:F(C);B
      (Listing for FILE2)
4 10 COMMON A,CUST$,B
   20 A = 10 : CUST$ = "MADELAIN" : B = 20
   30 CHAIN "FILE2"
   RUN
   MADELAIN 10 0 0
```

Notice in the above example that Vectra BASIC prints the value for the variable **B** as 0. Since the **COMMON** statement for **FILE2** omitted the variable **B**, Vectra BASIC assigns a value of zero to **B**.

**Note**

If you plan to compile your program, see the BASIC compiler manual for differences between the compile and interpretive versions of this statement.

**CONT Command**

**Format:**

CONT

**Purpose:**

Continues program execution after execution was suspended by either your typing [CTRL] Break or the program encountering a **STOP** or **END** statement.

**Remarks:**

You enter this command in Direct Mode.

Execution resumes at the point where the break occurred. If the break occurred after a prompt from an **INPUT** statement, execution continues by reprinting the prompt (?) or *prompt string*.

You normally use the **CONT** statement in conjunction with the **STOP** statement to debug a program. After execution stops, you may examine intermediate values by using Direct Mode statements. You may resume execution with the **CONT** statement (which continues with the next executable statement) or the Direct Mode **GOTO** statement (which continues execution at the specified line number).

You may also use **CONT** to resume execution after Vectra BASIC suspends execution upon its detecting an error condition. You may not use **CONT** to resume execution if you have modified the program (through edit commands) during the break, although you can use Direct Mode statements to alter the values in variables and then use **CONT** to resume execution.

**Example:**

The following program and interactive session illustrates how you might use the `CDNT` statement:

```
10 INPUT "ENTER PRICE", AMOUNT
20 IF AMOUNT < 20 THEN SURCHG=11
30 STOP
40 TOTAL = AMOUNT + SURCHG
50 PRINT TOTAL
RUN
ENTER PRICE
```

(you type 15 [Enter])

```
Break in 30
01
```

(you type PRINT SURCHG [Enter])

```
1
01
```

(you type CDNT [Enter])

```
16
01
```

For more information, see the `STOP` statement.

**Note**

The BASIC compiler offers no support for this command.

## COS Function

**Format:** `COS(x)`

**Action:** Returns the cosine of  $x$ , where  $x$  is given in radians.

**Note**

To convert degrees to radians, multiply the angle by  $\pi/180$ , where  $\pi \approx 3.141593$

Vectra BASIC evaluates `COS` in single-precision arithmetic.

To achieve a double-precision result,  $x$  must be defined as a double-precision variable and Vectra BASIC must be invoked with the `/D` switch, or the results of the function must be stored in a double-precision variable, as in `J# = COS(x)`.

**Example:**

```
10 X = 2 * COS(.4)
20 PRINT X
RUN
1.842122
01
```

## CSNG Function

**Format:** CSNG(X)

**Action:** Converts X to a single-precision number.

See the CINT and COBL functions for converting numbers to the integer and double-precision data types.

**Example:**

```
10 A# = 975.342124#
20 PRINT A#: CSNG(A#)
RUN
975.342124 975.3421
OK
```

## CSRLIN Function

**Format:** CSRLIN

**Action:** Returns the row position of the alphanumeric cursor.

The row position may range from 1 to 24 when the function key menu is displayed.

**Note**

With the function keys turned off (see the KEY statement), it is possible to use the LOCATE statement to print in the 25th row. CSRLIN will only return 25 in a statement such as LOCATE 25,5:PRINT CSRLIN.

You must use the POS function to return the current column position.

**Example:**

```
4000 REM Error-handling routine
4010 X = CSRLIN 'Record current line
4020 Y = POS(0) 'Record current column
4030 LOCATE 14,1 'Print message
4040 PRINT "ERROR ":ERR;" ENCOUNTERED".
4050 INPUT "PRESS RETURN TO CONTINUE". A#
4060 LOCATE X,Y 'Restore cursor's original position
4070 RESUME NEXT
```

## CVI, CVS, CVD Functions

### Format:

CVI(2-byte string)  
CVS(4-byte string)  
CVD(8-byte string)

### Action:

Converts string values to numeric values.

Random-access disc files store numeric values as strings. Therefore, when you read values from a random disc file, you must convert the strings into numbers.

CVI converts a 2-byte string to an integer.

CVS converts a 4-byte string to a single-precision number.

CVD converts an 8-byte string to a double-precision number.

See also MKI\$, MKS\$, MKD\$.

### Example:

```
70 FIELD #1, 4 AS N$, 12 AS B$, 2 AS A$  
80 GET #1  
90 CODE = CVS(N$)  
100 AGE$ = CVI(A$)
```

## DATA Statement

### Format:

DATA *constant 1, constant 1*

### Purpose:

Stores information (that is, numeric or string constants) for later access by a program's READ statements.

### Remarks:

*constant* may be a numeric or string constant.

Numeric constants may assume either an integer, fixed-point, or floating-point format. Numeric expressions are illegal.

You must place quotation marks around a string constant only if the string contains embedded commas or colons, or if it has significant leading or trailing spaces. Otherwise, you may omit the quotation marks.

DATA statements are nonexecutable. You may place them anywhere within the program.

A DATA statement may contain as many constants as you may fit on the input line. You must separate the DATA items by commas. Spaces before or after a comma are ignored.

A program's READ statements access the DATA statements in sequential order (by line number). Therefore, you may envision the data to be a continuous list of items, regardless of how many items are on a line or where the lines occur within the program.

The variable type given in the READ statement must agree with the corresponding constant in the DATA statement or a **Type mismatch** error occurs.

You may reread the information stored in a DATA statement by using the **RESTORE** statement.

## CVI, CVS, CVD Functions

### Format:

CVI(2-byte string)  
CVS(4-byte string)  
CVD(8-byte string)

### Action:

Converts string values to numeric values.

Random-access disc files store numeric values as strings. Therefore, when you read values from a random disc file, you must convert the strings into numbers.

CVI converts a 2-byte string to an integer.

CVS converts a 4-byte string to a single-precision number.

CVD converts an 8-byte string to a double-precision number.

See also MKI\$, MKS\$, MKD\$.

### Example:

```
70 FIELD #1, 4 AS N$, 12 AS B$, 2 AS A$  
80 GET #1  
90 CODE = CVS(N$)  
100 AGE$ = CVI(A$)
```

## DATA Statement

### Format:

DATA *constant* [, *constant*]

### Purpose:

Stores information (that is, numeric or string constants) for later access by a program's READ statements.

### Remarks:

*constant* may be a numeric or string constant.

Numeric constants may assume either an integer, fixed-point, or floating-point format. Numeric expressions are illegal.

You must place quotation marks around a string constant only if the string contains embedded commas or colons, or if it has significant leading or trailing spaces. Otherwise, you may omit the quotation marks.

DATA statements are nonexecutable. You may place them anywhere within the program.

A DATA statement may contain as many constants as you may fit on the input line. You must separate the DATA items by commas. Spaces before or after a comma are ignored.

A program's READ statements access the DATA statements in sequential order (by line number). Therefore, you may envision the data to be a continuous list of items, regardless of how many items are on a line or where the lines occur within the program.

The variable type given in the READ statement must agree with the corresponding constant in the DATA statement or a Type mismatch error occurs.

You may reread the information stored in a DATA statement by using the RESTORE statement.

**Example:**

```
10 DATA 80, 90, tonight, " dinner", 25
20 FOR I = 1 TO 5
30 READ A$
40 PRINT A$; " ";
50 NEXT I
60 END
RUN
80 90 tonight dinner 25
Ok
```

**DATE\$ Function**

**Format:** DATE\$

**Action:** Retrieves the current system date.

The date is originally set to the MS-DOS system date when Vectra BASIC is invoked.

When the date is reset with the DATE\$ statement, the system date is reset. The DATE\$ function fetches the date from the system date.

The DATE\$ function returns a 10-character string in the form:

*mm-dd-yyyy*

where:

*mm* is the month of the year. Values range from 01 to 12.

*dd* is the day of the month. Values range from 01 to 31.

*yyyy* is the year. Values range from 1980 to 2099.

**Example:**

```
PRINT DATE$
02-27-1984
Ok
```

## DATE\$ Statement

**Format:** DATE\$ = string

**Purpose:** Sets the current date for use by the DATE\$ function

**Remarks:** string represents the current date. You may enter it in one of the following forms:

mm-dd-yy  
mm-dd-yyyy  
mmdd/yy  
mmdd/yyyy

where:

mm is the month of the year. Values range from 01 to 12.

dd is the day of the month. Values range from 01 to 31.

yy or yyyy is the year. Dates may be set for the years 1980-2099. When you include only two digits, Vectra BASIC assumes 19 for the first two digits for the years 80-99 and 20 for the first two digits for years 00-77.

Attempting to set the date to a year before 1980 produces an illegal function call error.

You may omit leading zeroes in any of the fields.

### Example:

This example shows three different forms for entering the year. The final example omits leading zeroes for the month, day and year.

```
DATE$ = "01-01-1984"  
OK  
PRINT DATE$  
01-01-1984  
OK  
DATE$ = "02-27-84"  
OK  
PRINT DATE$  
02-27-1984  
OK  
DATE$ = "9-8-7"  
OK  
PRINT DATE$  
09-08-2007  
OK
```

## DEF FN Statement

**Format:** DEF FN *name* [(*parameter*1,...)] *definition*

**Purpose:** Names and defines a function which the user writes.

**Remarks:** *name* must be a legal variable name. This name, preceded by the letters FN, becomes the name of the function.

*parameter* is a variable name in the function definition that Vectra BASIC replaces with a value when the function is called. You must separate multiple parameters with commas.

*definition* is an expression that performs the operation of the function. You must limit the definition to one line (255 characters). Variable names that appear in this expression serve only as formal parameters to define the function. They have no effect on program variables that have the same name. A variable name used within the function definition might appear as a *parameter*. If it is a *parameter*, Vectra BASIC supplies its value when the function is called. Otherwise, Vectra BASIC uses the variable's current value.

The parameter variables correspond on a one-to-one basis to the argument variables or values that are given in the function call.

User-defined functions may be numeric or string. When the function name contains a type declaration character (% , ! or #), the value of the expression is forced to that type before Vectra BASIC returns the result to the calling statement. When you omit the type declaration character, Vectra BASIC considers the result to be a single-precision value. When a type is specified in the function name and the argument type differs, a **Type mismatch** error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an **Undefined user function** error occurs.

The DEF FN statement is illegal when you are using the Vectra BASIC interpreter in Direct Mode.

**Examples:** If a program contains the following lines:

```
30 VALUE(1) = A+Y/F-D
...
80 VALUE(1) = B+Y/F-E
...
200 VALUE(1) = C+Y/F-G
```

Then defining a function such as:

```
10 DEF FNNUM(S,T) = S+Y/F-T
```

simplifies the program to:

```
30 VALUE(1) = FNNUM(A,D)
...
80 VALUE(1) = FNNUM(B,E)
...
200 VALUE(1) = FNNUM(C,G)
```

The next example defines a different multiplication operation.

```
10 DEF FNMULT(I,J) = I*J*(1+2)*J*(1+3)*J
20 I = 2 : J = 3
30 A = FNMULT(I,J)
40 B = FNMULT(3,4)
50 PRINT A, B
RUN
42 156
OK
```

## DEF SEG Statement

**Format:** DEF SEG (*address*)

**Purpose:**

Assigns the current "segment" for storage. A subsequent BLOAD, BSAVE, CALL, CALLS, POKE, PEEK, or USR instruction defines the actual physical address that it requires as an offset into this segment.

**Remarks:**

*address* is a numeric expression that returns an unsigned integer which may range between 0 and 65535.

- If a legal function call. Under these circumstances, any previous value remains in effect.

Vectra BASIC saves the address you specify for use as the segment needed by a BLOAD, BSAVE, CALL, CALLS, POKE, PEEK, or USR instruction.

When you give an address, you should ensure that it is based on a 16-byte boundary. The value is multiplied by 16 (in binary, it is shifted left by 4 bits; in hex, a 0 is added) to form the segment address for the subsequent operation. Vectra BASIC does not check the validity of the specified address.

When you omit the *address* parameter, Vectra BASIC sets the segment address to that of the Vectra BASIC Data Segment (DS). This is the setting for the current segment when you initialize Vectra BASIC.

**Note**

You must separate DEF and SEG with a space. Otherwise, Vectra BASIC interprets the statement:

```
DEFSEG = 1000
```

as "assign the value of 1000 to the variable DEFSEG".

**Example:**

This example sets the segment address to &HB800. Later, a second statement (with no specified address) restores the address to the Data Segment (DS):

```
10 DEF SEG = &HB800
...
90 DEF SEG
```

## DEF USR Statement

**Format:** DEF USR (*digit*) = *offset*

**Purpose:** Gives the starting address of an assembly-language subroutine.

**Remarks:** *digit* may be any integer from 0 to 9. The digit corresponds to the number of the USR routine that you are specifying. When you omit the *digit* parameter, Vectra BASIC assumes the reference is to USR0.

*offset* is an integer expression whose value may range from 0 to 65535. Vectra BASIC adds *offset* to the value of the current storage segment set by a DEF SEG statement to get the actual starting address of the USR routine. (See Appendix C for information about assembly-language subroutines.)

DEF USR lets the programmer define starting addresses for user-defined assembly language functions that are called from Vectra BASIC programs. You must use this statement to set the starting address prior to its actual use.

A maximum of 10 user-defined functions are available for use at any given time. The routines are identified as USR0 to USR9. When you need access to more subroutines, you can use multiple DEF USR statements to redefine a subroutine's starting address. However, Vectra BASIC only saves the last-executed value as the offset for that subroutine.

### Note

The CALL statement is the preferred way of calling subroutines. You should avoid using the USR statement.

**Example:** This example calls the user function at the Data Segment relative memory location 24000:

```
200 DEF SEG = 0
210 DEF USR0 = 24000
220 X = USR0 (Y*2/2.89)
```

**DEFINT/SNG/DBL/STR Statements**

**Format:**

```
DEFINT letter (-letter) (,letter (-letter)) ...  
DEFSNG letter (-letter) (,letter (-letter)) ...  
DEFDBL letter (-letter) (,letter (-letter)) ...  
DEFSTR letter (-letter) (,letter (-letter)) ...
```

**Purpose:**

Declares that Vectra BASIC should automatically treat certain variable names as integer, single-precision, double-precision, or string variables, respectively.

**Remarks:**

*letter* is a letter of the English alphabet (A-Z).

Vectra BASIC considers any variable names beginning with the specified letter(s) to be of the requested type. However, when assigning variable types, Vectra BASIC always gives precedence to a type declaration character (**I**, **S**, **D**, or **\$**) over an assignment set by a *DEF type* statement.

In the following example, Vectra BASIC prints the variable **C** as an integer because of the type declaration character (**I**), even though **C** is within the range of the **DEFDBL** declaration.

```
10 DEFDBL B-D  
20 D = S.2D*17 : CX = 20.2  
30 PRINT D,CX  
RUN  
S.2D*17      20  
OK
```

When you use these statements, you should place them at the beginning of a program. (Vectra BASIC must execute the *DEF type* statement before you use any variables that it declares.)

If a program contains no type declaration statements, Vectra BASIC assumes that any variable without a declaration character is a single-precision variable.

**Examples:**

The first example defines all variables that begin with either the letter **L**, **M**, **N**, **O**, or **P** to be double-precision variables:

```
10 DEFDBL L-P
```

The next statement defines all variables that begin with the letter **A** to be string variables:

```
10 DEFSTR A
```

The last example defines all variables that begin with either the letter **I**, **J**, **K**, **L**, **M**, **N**, **U**, **X**, **Y**, or **Z** to be integer variables:

```
10 DEFINT I-N,U-Z
```

**Note**

If you plan to compile your program, see the BASIC compiler manual for differences between the interpretive and compiled version of this statement.

## DELETE Command

- Format:**     DELETE (*start line*) [(*end line*)]
- Purpose:**     Deletes the specified line(s) from a Vectra BASIC program.
- Remarks:**   *start line* is the number for the first line you want to delete.  
              *end line* is the number for the last line you want to delete.

**Note**        When you omit both *line* parameters, Vectra BASIC deletes the entire program.

You may use a period (.) in place of a line number when you want to delete the current line.

If Vectra BASIC fails to find the line number you supplied, it returns an illegal function call.

Vectra BASIC always returns control to the command level after the DELETE command executes.

### Examples:

- The first example only deletes line 40:
- ```
DELETE 40
```
- The next statement deletes from line 40 to the end of the program:
- ```
DELETE 40 -
```
- The next statement deletes from the beginning of the program through line 40:
- ```
DELETE -40
```
- The last example deletes all lines between 40 and 80, inclusively:
- ```
DELETE 40-80
```

### Note

The BASIC compiler offers no support for this command.

## DIM Statement

**Format:**

`DIM arrayname (subscripts) [, arrayname (subscripts)] ...`

**Purpose:**

Sets the maximum values for the subscripts of an array variable, allocates the necessary storage, and initializes the elements of the array to zero or null.

**Remarks:**

*arrayname* is a variable that names the array.

*subscripts* is a list of numeric expressions, separated by commas, that define the array's dimensions.

When you fail to dimension an array with the DIM statement, Vectra BASIC assumes the maximum subscript is 10. If you subsequently use a subscript that exceeds this number, a **Subscript out of range** error occurs.

If you need to redimension an array, you should first remove it using the ERASE statement. (The CLEAR statement removes arrays, but it also resets all other variables in a program.) If you try to redimension an array without using ERASE or CLEAR, a **Duplicate Definition** error occurs.

The minimum value for an array subscript is zero unless you use the OPTION BASE statement to change it to one.

The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767. However, these values are theoretical limits as both values are limited by the size of memory and the number of characters that you can enter on the input line.

The DIM statement sets all elements of numeric arrays to an initial value of zero and all elements of string arrays to the null string.

**Example:**

```
10 DIM ID(20)
20 FOR I = 0 TO 20
30   READ ID(I)
40 NEXT I
```

**Note**

If you plan to compile your program, see the BASIC compiler manual for differences between the compiled and interpretive version of the DIM statement.

## DRAW Statement

**Format:**

DRAW *string*

**Purpose:**

Draws the specified object. This command is only valid in a graphics mode.

**Remarks:**

*string* is made up of the following commands where *n* shows the distance, in pixels, traveled from the Current Graphics Position (CGP). (The Current Graphics Position is usually the coordinate of the last graphic point that a LINE or PSET statement places on the screen. When a program is first run, the CGP defaults to the center of the screen.)

When you omit the *n* parameter, Vectra BASIC moves one pixel in the indicated direction.

U [ <i>n</i> ]	Move up
D [ <i>n</i> ]	Move down
L [ <i>n</i> ]	Move left
R [ <i>n</i> ]	Move right
E [ <i>n</i> ]	Move diagonally up and right
F [ <i>n</i> ]	Move diagonally down and right
G [ <i>n</i> ]	Move diagonally down and left
H [ <i>n</i> ]	Move diagonally up and left
M <i>x,y</i>	Move absolute to point ( <i>x,y</i> )
M ± <i>x</i> , ± <i>y</i>	Move relative to current point

**Note**

When you precede *x* by a plus (+) or minus (-), Vectra BASIC offsets the point relative to the Current Graphics Position.

The DIM statement sets all elements of numeric arrays to an initial value of zero and all elements of string arrays to the null string.

**Example:**

```
10 DIM ID(20)
20 FOR I = 0 TO 20
30   READ ID(I)
40 NEXT I
```

**Note**

If you plan to compile your program, see the BASIC compiler manual for differences between the compiled and interpretive version of the DIM statement.

## DRAW Statement

**Format:**

DRAW *string*

**Purpose:**

Draws the specified object. This command is only valid in a graphics mode.

**Remarks:**

*string* is made up of the following commands where *n* shows the distance, in pixels, traveled from the Current Graphics Position (CGP). (The Current Graphics Position is usually the coordinate of the last graphic point that a LINE or PSET statement places on the screen. When a program is first run, the CGP defaults to the center of the screen.)

When you omit the *n* parameter, Vectra BASIC moves one pixel in the indicated direction.

U [ <i>n</i> ]	Move up
D [ <i>n</i> ]	Move down
L [ <i>n</i> ]	Move left
R [ <i>n</i> ]	Move right
E [ <i>n</i> ]	Move diagonally up and right
F [ <i>n</i> ]	Move diagonally down and right
G [ <i>n</i> ]	Move diagonally down and left
H [ <i>n</i> ]	Move diagonally up and left
M <i>x,y</i>	Move absolute to point ( <i>x,y</i> )
M ± <i>x</i> , ± <i>y</i>	Move relative to current point

**Note**

When you precede *x* by a plus (+) or minus (-), Vectra BASIC offsets the point relative to the Current Graphics Position.

You may precede the above movement commands with either of these two commands:

**B**

Move but do not draw any points  
Move but return to the original position

**N**

You may also use the following commands for special effects:

**A*n***

Set angle *n* where:

*n* = 0 means 0 degrees

*n* = 1 means 90 degrees

*n* = 2 means 180 degrees

*n* = 3 means 270 degrees

**TA*n***

Turn Angle *n*, where *n* is an angle that ranges between -360 and 360 degrees. Positive values imply a counterclockwise rotation while negative values imply a clockwise rotation. When *n* is outside this range, an illegal function call occurs.

**CO*n***

Sets the color for subsequent DRAW commands to *n*.

In medium resolution, *n* may range from 0 to 3, selecting a color from the palette chosen in a COLOR statement. In high resolution, 0 selects the background color and 1 selects the foreground color.

S*n*

Sets the scale factor. The value for *n* may range from 1 through 255, and *n* divided by 4 yields the scale factor. For example, *n* = 1 gives a scale factor of 1/4. The default value of 4 for *n* gives a scale factor of 1.

This command, in conjunction with a "Move" command (that is, U, D, L, R, E, F, G, H, and M) gives the actual distance moved. For example, a value of 8 for *n* gives a scale factor of 2. Therefore, U3 would move the cursor up 6 pixels.

X *string*

This parameter executes a substring within a string. You may have one string execute another string which executes a third, and so on. For example, you may use this command to separate part of an object from the entire object, then move it independently.

P *paint*, *boundary*

Fills a figure (or the screen) with the color *paint*, up to a line of color *boundary*.

*paint* and *boundary* may range from 0 to 3 in medium resolution. They reference the colors of the palette selected by the COLOR statement.

In high resolution, *paint* and *boundary* may range from 0 to 255. 0 and 2 produce the background color; all other values produce the foreground color set by a COLOR statement.

In all of these commands, the numeric arguments (*n*, *x*, and *y*) can be constants or variables. For example, 7 is a constant. You use variable names by writing:

```
*variable;
```

where *variable* is the name of a numeric variable. Remember, when you use a variable in this way, you must precede the variable name with an equal sign and follow it with a semicolon. For example, the relative Move command M-X, Y becomes M-*var1*;; \**var2*;

### Note

You can use the function call, **VARPTR(*variable*)**, instead of **\**variable***!. Out of range coordinates give an illegal function call.

The DRAW statement uses absolute addressing; it is not influenced by world coordinates.

**Examples:**

```
90 SCREEN 1
100 REM Draw a triangle
110 DRAW "E15 F15 L30"
120 REM Draw a box
130 US="U30;" : DS="D30;" : LS="L40;" : RS="R40;"
140 BOX$=US + RS + DS + LS
150 DRAW "XB0X$;"
```

You could also draw the same box by using the **x** subcommand and replacing lines 140 and 150 with the following line:

```
140 DRAW "XU$; XR$; XD$; XL$;"
```

The next example draws the spokes of a wheel by using the **TA** (Turn Angle) option:

```
10 SCREEN 1
20 CIRCLE (256,60),50
30 FOR D = 0 TO 360 STEP 10
40 DRAW "TA:D;NU50"
50 NEXT D
```

The following program draws a box then uses the *paint* option to fill its interior:

```
10 SCREEN 1
20 DRAW "U50R50DS0LS0" '50 pixels to a side
30 DRAW "BE10" 'Move inside area
40 DRAW "P2,3" 'Paint interior
```

**EDIT Command**

**Format:** EDIT line  
EDIT .

**Purpose:** Displays a line for editing.

**Remarks:** The EDIT command displays the specified line, places the cursor on the first character of the line, and then waits for your edit changes. You may then modify the line with any of the techniques presented in Chapter 1.

When you specify a line number, the EDIT command edits that line. If no such line exists, an *undefined line number* error occurs.

When you enter EDIT ., the EDIT command edits the last line that you typed, the last line that a LIST statement displayed, or the last line that an error message referenced.

**Examples:** Both of the following groups of commands display Line 10 for editing:

```
EDIT 10
LIST 10
EDIT .
```

**Note** The BASIC compiler offers no support for this command.

## END Statement

### Format:

END

### Purpose:

Stops program execution, closes all files, and returns control to the command level.

### Remarks:

You may place END statements anywhere in a program to end execution. The END statement at the end of a program, however, is optional. When you omit it, execution stops after the last line in the program executes, without closing open files.

The END statement differs from the STOP statement in two important ways:

- END closes all files
- END terminates the program without printing a Break message

Vectra BASIC always returns control to the command level after an END statement executes.

### Example:

This program segment tests to see if more data exists. END statements terminate the program when no data exists and prevent program flow from falling into the subroutine section:

```
S20 IF EOF(1) THEN END ELSE GOTO 200
      .
850 END
1000 REM THE FOLLOWING SECTION CONTAINS
1010 REM THE INPUT SUBROUTINES
```

### Note

If you plan to compile your program, refer to the BASIC Compiler Manual for programming differences when using the END statement.

## ENVIRON Statement

### Format:

ENVIRON string

### Purpose:

Modifies a parameter in MS-DOS's Environment String Table

### Remarks:

string is a string expression, in the form parameter+text or parameter text. Everything to the left of the equal sign or space is assumed to be a parameter, and everything to the right is assumed to be text.

If the parameter already exists in the Environment String Table, the ENVIRON statement deletes the old definition, and add the new definition to the end of the table. If the parameter does not already exist in the table, it is appended to the table.

To remove a parameter from the table, use the form parameter=.

### Note

See the ENVIRON and the SHELL command for more information. Also see the SET command in the *HP Vectra MS-DOS User's Guide*.

This statement can be used to change the path for a SHELL command or to pass parameters to a child process by inventing a new environment parameter.

When Vectra BASIC is invoked, it reads the Environment String Table, and sets a fixed length for the table. There is very little extra space, and it is not possible to increase that space from within Vectra BASIC. If you issue an ENVIRON statement with an argument that exceeds this storage space, an Out of memory error message results.

If you need to create a long pathname (or other lengthy parameter) from a Vectra BASIC program, you can save space in the table by inserting a dummy parameter into the table from DOS, then deleting that parameter.

For instance, you might issue this command from DOS (or in an AUTOEXEC.BAT file) before invoking Vectra BASIC:

```
SET DUMMY=this is just a very long dummy string
to make room in BASIC
```

Your Vectra BASIC program could contain the lines:

```
ENVIRON "DUMMY";
ENVIRON "PATH=\\ACCOUNTS\\PAYABLES\\MARCH\\POSTINGS"
```

**Examples:** The following MS-DOS command creates a default "PATH" to the root directory on disc A:

```
PATH=A:
```

From Vectra BASIC, you can change that path by:

```
ENVIRON "PATH=A:BOOKS\\MAY"
```

This command adds a new parameter to the table:

```
ENVIRON "SESAME-PLAN"
```

## ENVIRON\$ Function

**Format:** ENVIRON\$(string,parameter*n*)

**Action:** Retrieves a parameter string from Vectra BASIC's Environment String Table.

*string parameter* is the name of the parameter in the table. This form of the ENVIRON\$ function returns a string containing the definition that matches the parameter. If no parameter in the table matches *string parameter*, or if there is no text after the matching entry, this function returns the null string.

*n* can be an integer between 0 and 255. This form of the ENVIRON\$ function returns the *n*th line from the table. It returns the entire line, that is *parameter string*. If there are less than *n* lines in the table, this command returns the null string.

The string form of the ENVIRON\$ function pays attention to case in the parameter names. You cannot use "path" in lower case to return the definition for "PATH" entered in upper case, even through both Vectra BASIC and DOS ignore case in commands.

**Examples:**

The first example uses a FOR...NEXT loop to print the entire contents of the table. Since there are only 2 items, it prints 3 blank lines. The last example also returns the null string, since "Path" does not equal "PATH".

```
0k
FOR N=1 TO 5:PRINT ENVIRON$(CJ):NEXT
COMSPEC=C:\COMMAND.COM
PATH=A:\BOOKS

0k
PRINT ENVIRON$ "PATH"
A:\BOOKS
0k
PRINT ENVIRON$(2)
PATH=A:\BOOKS
0k
PRINT ENVIRON "Path"
0k
```

**EOF Function**

**Format:**

EOF(*filename*)

**Action:**

For sequential files, the EOF function returns true (— 1) when no more data exists in the file. Vectra BASIC considers the file empty if the next input operation (for example, INPUT or LINE INPUT) would cause an input past end error. Using the EOF function to test for the end-of-file while inputting information avoids such errors.

For random-access files, EOF returns true (— 1) if the most recently executed GET statement attempts to read beyond the end-of-file.

Because Vectra BASIC allocates 128 bytes to a file at a time, it is possible that EOF will not accurately detect the end of a random-access file that was opened with a record length of less than 128 bytes. For example, if you open a file with a record length of 64 bytes and you write one record to the file (that is, PUT #1, 1), EOF returns false if a GET statement is attempted on the file's record (for example, GET #1, 1). This occurs even though the record has not actually been written.

**Example:**

This sample program lists the titles of the books cataloged in the file LIBRARY.DAT. It also counts the books in the library by counting the number of records that it reads from LIBRARY.DAT before it encounters the end-of-file.

Each record of LIBRARY.DAT contains information on one book. The record length is 128 bytes. The first 35 bytes contain the title of the book. The remaining 93 bytes contain additional information such as the author, publisher, print date, and so on.

```

10 REM      Open the library catalog file.
20 REM      LIBRARY.DAT.
30 OPEN "R",I,"LIBRARY.DAT"
40 REM      The first 35 bytes of the
50 REM      record contain the title.
60 REM      the remaining 93 bytes contain
70 REM      additional information that
80 REM      this program does not use.
90 FIELD 1, 35 AS TITLE$, 93 AS G$
100 FIELD 1, 35 AS TITLE$, 93 AS G$
110 REM
120 REM      Initialize the number of books seen.
130 REM
140 NBOOKS = 0
150 REM      Attempt to fetch the next record.
160 REM      Note that the record number
170 REM      of GET isn't specified
180 REM      so the next record of the file
190 REM      is fetched.
200 GET 1
210 REM
220 REM      Is this the end of the file?
230 REM
240 IF EOF(1) THEN 1000
250 REM      If no: increment the count of books.
260 REM      print the current title, and
270 REM      loop back to read the next record.
280 REM
290 NBOOKS = NBOOKS + 1
300 PRINT TITLE$
310 GOTO 200
1000 REM      Control passes here when the end of
1010 REM      file has been reached, so:
1020 REM      print a blank line and the number of
1030 REM      books, close the file, and terminate
1040 REM      the program.
1050 PRINT "There are "; NBOOKS; " books in ";
1060 PRINT "your library."
1070 CLOSE
1080 END

```

## EOF Function (for COM files)

The following discussion pertains to communications files:

**Format:** EOF (*filename*)

**Action:** Tells whether the input queue is empty.

The end-of-file condition depends on the mode (ASCII or binary) in which the device was opened.

In binary mode, EOF is true (—1) when the input queue is empty (LOC(n)=0). EOF becomes false (0) when the queue is not empty.

In ASCII mode, EOF is false until a Control-Z is received. From then on, it will remain true until the device is closed.

### Example:

```

10 OPEN "COM2:" AS #1
20 C=0
30 IF EOF (1) THEN 100
40 A$ = INPUT$(LOC(1),#1)
50 C=C+1 : GO TO 30

```

## ERASE Statement

**Format:** ERASE *arrayname* [, *arrayname*] ...

**Purpose:** Deletes the named arrays from memory and reclaims their storage space.

**Remarks:** *arrayname* names the array that you want to delete.

After you delete an array, you may redimension that array or use the previously allocated array space for another purpose.

Attempting to redimension an array without first erasing it causes a `Duplicate Definition` error.

**Example:** 450 ERASE ID, STATS  
460 DIM ID(99)

### Note

The BASIC compiler offers no support for this statement.

## ERDEV and ERDEV\$ Variables

**Format:** ERDEV  
ERDEV\$

ERDEV contains the error code returned by the last device to declare an error. ERDEV\$ contains the name of the device driver which generated the error.

When DOS detects an error on a device, ERDEV holds the Interrupt 24H codes that the driver of that device generated.

ERDEV\$ will contain the name of the device which generated the error. If the device which generated the error was a character device, such as a printer, ERDEV\$ will contain the name of the device, for example, LPT1. Otherwise, ERDEV\$ will contain the 2-character block device name, such as A: or B: for a disc drive.

### Note

These variables are "read-only". You can print or read the contents, but you cannot store values in these variables.

See the *HP Vectra MS-DOS Programmers Reference* for more information about Interrupt 24H codes.

### Examples:

This portion of an error trapping routine checks to see if the error was caused by an open disk drive door.

```
2020 IF ERDEV=2 THEN PRINT "PLEASE CLOSE  
      THE DOOR ON DRIVE " ERDEV;
```

If you have installed a device driver called "MYLPT2" that sends an error code 9 when the printer runs out of paper, this statement:

```
PRINT ERDEV, ERDEV;
```

prints

```
9 MYLPT2
```

## ERR and ERL Variables

### Format:

```
ERR  
ERL
```

### Action:

When Vectra BASIC enters an error-handling routine, the variable **ERR** contains the error code for the error, and the variable **ERL** contains the line number of the line in which Vectra BASIC detected the error.

You normally use these variables in **IF...THEN** statements to direct program flow in the error trap routine.

When the statement causing the error was a Direct Mode statement, **ERL** contains the value 65535. To test if an error occurred in a Direct Mode statement requires the following statement:

```
IF ERL = 65535 THEN...
```

You may also test for other error conditions by using the following statements:

```
IF ERR = error.code THEN
```

or

```
IF ERL = line# THEN
```

You could also enter the previous statement as:

```
IF line# = ERL THEN
```

However, when *line#* appears on the left side of the equal sign, the **RENUM** command fails to adjust the value for *line#* if its value changes while resequencing the program.

Caution

Numeric constants following an ERL variable in a given expression may be treated as line references and thus modified by a RENUM statement. To avoid this problem, you should use statements similar to these:

L • ERL : PRINT L/10

rather than this statement:

PRINT ERL/10

ERL and ERR are variables that Vectra BASIC reserves for its use. Therefore, Vectra BASIC prevents you from assigning values to these variables. For example, the following assignment is illegal:

LET ERR = 65535

Appendix A lists the Vectra BASIC error codes.

ERROR Statement

Format:

ERROR *number*

Purpose:

Either simulates the occurrence of a Vectra BASIC error or allows you to define error codes.

Remarks:

*number* must be an integer expression between 0 and 255. When the value of *number* is equal to a Vectra BASIC error message, the ERROR statement simulates the occurrence of that error (which includes the printing of the corresponding error message). (See the first example.)

To define your own error code, select a value that is greater than those used by the Vectra BASIC error codes. (We recommend that you use the highest available values, for example numbers over 200, so your program can maintain compatibility if Vectra BASIC adds more error codes in later versions of this package.) This user-defined error code may then be conveniently handled in an error-trap routine. (See the last example.)

When an ERROR statement specifies a code for an error message that is undefined, Vectra BASIC responds with the message `Unprintable error`.

Executing an ERROR statement for which no error-trap routine exists prints an error message and halts execution.

### Examples:

```
10 S = 10
20 T = S
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

If you are using the Vectra BASIC interpreter in Direct Mode, you may enter an error number at the Ok prompt.

For example, if you enter:

```
ERROR 15
```

Vectra BASIC responds:

```
String too long
Ok
```

The last example shows how you may define your own error codes.

```
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET?"; WAGER
130 IF WAGER > 5000 THEN ERROR 210
...
400 IF ERR=210 THEN PRINT "HOUSE LIMIT IS 5000"
410 IF ERR=130 THEN RESUME 120
```

## EXP Function

**Format:** EXP(*x*)

### Action:

Returns  $e$  (where  $e = 2.71828...$ ) to the power of  $x$ . The number  $e$  is the base of the natural logarithms.

$x$  must be less than 88.02969.

The EXP function returns a single precision value unless the result is stored in a double precision variable; for example,  $J = \text{EXP}(x)$ ; or unless the /D switch was used when Vectra BASIC was invoked and a double precision variable was used as the argument; for example, `PRINT EXP(x#)`.

If EXP overflows, Vectra BASIC displays the Overflow error message, sets the result to machine infinity with the appropriate sign, and continues execution.

### Example:

```
10 X = 5
20 PRINT EXP (X-1)
RUN
54.59815
Ok
```

## FIELD Statement

### Format:

`FIELD ( * ) filename, field, width AS stringvar  
[, field, width AS stringvar], ...`

### Purpose:

Allocates space for variables in the random file buffer.

### Remarks:

Vectra BASIC reads and writes random files through a file buffer that holds the file record. You must assemble and disassemble this buffer into individual variables. Therefore, this requires your using the `FIELD` statement to specify the layout of the file buffer before you get data out of a random file buffer after a `GET`, or to enter data before a `PUT`.

*filenum* is the number you gave the file when you opened it.

*field, width* is the number of character positions that you want to allocate to *stringvar*. For example, the following statement allocates the first 20 positions (bytes) in the random file buffer to the string variable `CHNAME$`, the next 10 bytes to `ID$`, and the next 40 bytes to `ADDRESS$`:

```
FIELD #1, 20 AS CHNAME$, 10 AS ID$, 40 AS ADDRESS$
```

*stringvar* is a string variable that is used for random file access.

The `FIELD` statement is a template for formatting the random file buffer. It never places any data into the buffer. (See the `GET` and `LIST/RSET` statements for information on moving data into and out of the random file buffer.)

Numeric values are stored as strings in random files. The `PK$`, `MKS$`, and `MKD$` functions convert numeric values to strings. The `field` statement must reserve the proper number of bytes for each of the variable types:

Bytes	Variable type
2	integer
4	single precision
8	double precision

### Note

In some versions of BASIC, a `FIELD` statement remains in effect after a file is closed. In Vectra BASIC, if you `CLOSE` a file and then re-open it, you must issue the `FIELD` statement again. No warning is given if this is not done; however, your data is not written to disc by the `PUT` statement.

You may execute any number of `FIELD` statements for a given file. Once it executes, a `FIELD` statement remains in effect until the file is closed. Each new `FIELD` statement redefines the buffer from the first character position. This permits multiple field definitions for the same data.

The total number of bytes you allocate with a `FIELD` statement must not exceed the record length that you set when you opened the file. (When you omit specifying the length parameter, Vectra BASIC sets the record length to 128 bytes.) Attempting to allocate more bytes than the record can hold results in a `FIELD` overflow error.

If your definition of a record's layout requires more than 255 characters, you must divide the definition into two or more FIELD statements. For example:

```
10 OPEN "R", #1, "FILE", 120
20 FIELD #1, 2 AS ACODE$, 2 AS BCODE$, 4 AS ACTNM$,
   2 AS DCODE$, 6 AS CITY$, 10 AS LASTNAME$,
   2 AS ALTCODE$, 4 AS DFLAG$, 2 AS KYHUM$,
   8 AS BDAT$, 8 AS LDANDATE$, 2 AS PAYCODE$,
   5 AS PYMTGRD$, 5 AS CHECKHUM$
30 FIELD #1, 62 AS DUMMY$, 40 AS COMMENTS$,
   18 AS FRSTNAME$
```

In this example, DUMMY\$ is a string variable whose width is equal to the combined width of all the variables in the previous FIELD statement. It provides a way of skipping over the buffer space that you allocated to variables in the first FIELD statement.

- Never assign a LSET or RSET value to these dummy variables.

## Note

Be careful how you use a field variable name in an INPUT or LET statement. After you assign a variable name to a field, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable's name executes, the variable's pointer moves to string space and ceases to be in the file buffer.

## Example:

```
10 OPEN "R", #1, "FILE", 40
20 FIELD #1, 20 AS CUST$, 4 AS PRICE$, 16 AS CITY$
30 INPUT "CUSTOMER NUMBER", CODE$
40 INPUT "CUSTOMER NAME", CNAME$
50 INPUT "TOTAL ORDER": AMT
60 INPUT "CITY": TOWN$
70 LSET CUST$ = CNAME$
80 LSET PRICE$ = MK$(AMT)
90 LSET CITY$ = TOWN$
100 PUT #1, CODE$, CNAME$, TOWN$
110 GOTO 30
```

## FILES Command/Statement

**Format:** FILES (*filename*)

**Purpose:** Lists the names of the files that reside on the specified disc, or in the specified directory or subdirectory of the disc.

**Remarks:** *filename* is a string expression that contains the name of the file. It may contain an optional device designator and path.

If no drive designator is included in *filename*, Vectra BASIC uses the current drive. If no path is specified, Vectra BASIC searches the current directory for *filename*.

If *filename* is a literal, you must enclose the name in quotation marks.

*filename* is a string formula that may contain question marks (?) or asterisks (\*) as wild cards. A question mark matches any single character in the filename or extension. For example, CHAP? would match CHAP1, CHAP2, CHAP5, and so on. An asterisk matches one or more characters, beginning at that position. For example, CHAP\* not only matches all the files listed above but also matches CHAPTER, CHAPLAIN, CHAPEAU, and so on.

Omitting *filename* lists all the files in the current directory on the currently selected drive.

**Examples:**

This statement lists all the files on the current directory of the current disc:

```
FILES
```

The next statement lists all files with the BASIC file type extension (.BAS):

```
FILES "*.BAS"
```

This statement lists all the BASIC files with a PROG prefix and one trailing character, such as PROGS.BAS or PROG1.BAS:

```
FILES "PROG?.BAS"
```

The following example lists all the files in the directory SPORT\HELEN:

```
FILES "SPORT\HELEN\*"
```

**Note**

In the example above, the final backslash must be included, or you will be asking for the list of files named HELEN in \SPORT. The statement FILES "SPORT\HELEN" without the final backslash results in:

```
HELEN      <DIR>
```

The following example lists all the files with ".BAS" extenders in the directory SPORT\HELEN:

```
FILES "SPORT\HELEN\*.BAS"
```

---

## FIX Function

**Format:**        FIX(*x*)

**Action:**       Returns the truncated integer portion of *x*.

**FIX(*x*)** is equivalent to  $\text{SIGN}(x) * \text{INT}(\text{ABS}(x))$ . The major difference between **FIX** and **INT** is that **FIX** does not return the next lower number for negative *x*.

For example,

**FIX(-3.99)** returns -3

whereas

**INT(-3.99)** returns -4.

**Examples:**

```
PRINT FIX (58.75)
```

```
58
```

```
OK
```

```
PRINT FIX (-58.75)
```

```
-58
```

```
OK
```

## FOR...NEXT Statement

**Format:**

```
FOR variable = x TO y (STEP z)
    [loop statements]
NEXT [variable] [, variable] ...
```

**Purpose:** Loops through a series of statements a given number of times.

**Remarks:** *variable* serves as a counter.

*x*, *y*, and *z* are numeric expressions.

*x* is the initial value of the counter.

*y* is the final value of the counter.

*z* is the increment. When you omit this parameter, Vectra BASIC increments the count by one on each iteration through the loop.

If STEP is negative, the final value of the counter is set to be less than the initial value. Under these circumstances, Vectra BASIC decrements the counter on each iteration through the loop, and looping continues until the counter is less than the final value.

Vectra BASIC executes the program lines that follow the FOR statement until it encounters the NEXT statement. Vectra BASIC then increments the counter by the amount specified by STEP. It then checks to see if the value of the counter exceeds the final value (*y*). If it is not greater than the final value, Vectra BASIC branches back to the first statement within the loop and repeats the process. When the counter finally exceeds the final value, execution continues with the statement after the NEXT statement.

You may modify the value of *variable* from inside the loop. However, we do not recommend this practice.

If the initial value of the loop times the sign of the step exceeds the final value times the sign of the step, Vectra BASIC skips over the FOR...NEXT loop.

You may place a FOR...NEXT loop within the context of another FOR...NEXT loop. When you nest loops in this fashion, each loop must have a unique variable name for its counter. Furthermore, the NEXT statement for the inner loop must appear before the NEXT statement of the outer loop. When nested loops have the same end point, you may use a single NEXT statement which names each of the counters.

The variable name(s) in the NEXT statement are optional, except when ending nested loops with one NEXT statement.

If a NEXT statement is encountered before its corresponding FOR statement, Vectra BASIC displays a NEXT without FOR error and halts execution.

### Examples:

Although the following example modifies the loop's final value, it has no effect on program execution since Vectra BASIC calculates this value only once when it first enters the FOR statement:

```
10 K = 10
20 FOR I = 1 TO K STEP 2
30 PRINT
40 FOR J = 1 TO 3
50 K = K + 1
60 PRINT K;
70 NEXT J
80 NEXT I
90 END
RUN
11 12 13
14 15 16
17 18 19
20 21 22
23 24 25
OK
```

Vectra BASIC skips the FOR loop in the following example since the initial value of the loop exceeds the final value and a negative STEP doesn't appear:

```
10 J = 0
20 FOR I = 1 TO J
30 PRINT I
40 NEXT I
```

The loop in the next example executes ten times since Vectra BASIC always calculates the final value for the loop value before it sets the initial value.

### Note

Previous versions of BASIC set the initial value of the loop counter variable before setting the final value. Were this still true in the following example, the loop would have executed 6 times and not 10.

```
10 I = 5
20 FOR I = 1 TO I + 5
30 PRINT I;
40 NEXT
RUN
1 2 3 4 5 6 7 8 9 10
OK
```

In the statement,

```
FOR I = 45 TO 45.8 STEP 0.2
```

Vectra BASIC executes the loop four times; and not five times as you would expect. This results from the computer's attempt to represent decimal digits in a binary format.

On each iteration of the loop, the value for the counter takes on these values:

```
45
45.20000076293945
45.40000152587891
45.60000228881836
45.80000305175781
```

As the last value exceeds 45.8, the FOR loop terminates after the fourth iteration.

### Note

If you plan to compile your program, see the BASIC compiler manual for differences between the compiled and interpretive versions of this statement.

## FRE Function

### Format:

```
FRE (0)  
FRE (15)  
FRE (***)
```

### Action:

Returns the number of bytes of memory that are available for the user's program.

The FRE arguments are dummy arguments.

FRE (\*\*\* ) forces the system to reorganize the memory that Vectra BASIC uses, so no space is used by unreferenced variables. It then returns the total number of free bytes.

Normally, Vectra BASIC does not initiate memory consolidation until it uses its allotment of free memory.

Using FRE (\*\*\* ) periodically in your program will result in short delays for clean-up, rather than a possibly long delay if Vectra BASIC initiates this process.

### Example:

```
PRINT FRE (0)  
14542  
Ok
```

## GET Statement

### Format:

```
GET ( # ) filename ( , record )
```

### Purpose:

Reads a record from a random disc file into the random file buffer.

### Remarks:

*filename* is the number you gave the file when you opened it.

*record* identifies the record to be read. The value for *record* may range from 1 to 32767.

When you omit *record*, Vectra BASIC reads the next record, which followed the last GET, into the buffer.

### Note

After a GET statement, you may use the INPUT# statement and/or the LINE INPUT# statement to read characters from the random file buffer.

### Example:

```
10 OPEN "P", #1, "FILE", 40  
20 FIELD #1, 20 AS CUST#, 4 AS PRICES, 16 AS CITY$  
30 INPUT "ENTER CUSTOMER NUMBER"; CODE$  
40 IF CODE$ = 0 THEN END  
50 GET #1, CODE$  
60 PRINT CODE$  
70 PRINT USING "###.##"; CVS(PRICES)  
80 PRINT CITY$ : PRINT  
90 GOTO 30
```

## GET Statement (for graphics applications)

In addition to the standard GET statement, the graphics format of GET is:

**Format:**

GET (x1,y1)-(x2,y2), *arrayname*

**Purpose:**

Transfers graphics images from the screen into an array. Vectra BASIC reads the points of the image bound by the specified rectangle.

**Remarks:**

x1,y1 and x2,y2 are opposite corners of a rectangular area. You may give these coordinates in absolute or relative form.

*arrayname* is the name of the array where the image is stored. The dimensions must be large enough to hold the image. The array may be any type except string. However, unless the array is type integer, the contents of the array after a GET statement are meaningless if you try to interpret them directly.

In medium resolution, each pixel on the screen is represented by two bits of stored information; in high resolution, only one bit. Here are the formulas to compute the amount of storage needed:

Medium resolution:

$$\text{BYTES} = 4 * \text{INT}((X * 2 + 7) / 8) * Y$$

High resolution:

$$\text{BYTES} = 4 * \text{INT}((X * 7) / 8) * Y$$

X is the width and Y is the height of the image in pixels or dots. If you want to save an image that spans pixels 5 to 10, remember that this is 6 pixels, not 5. If the array size you specify is too small, the GET statement results in an illegal function call error message.

Once you've determined the number of bytes required to store the image, use the following divisors to determine the size of the array needed, depending on the type of array:

- 2 bytes for integers
- 4 bytes for single-precision numbers
- 8 bytes for double-precision numbers

If an image required 94 bytes, you would need to use one of these dimension statements:

Array type	Statement
integer	DIM A%(47)
single precision	DIM A!(24)
double precision	DIM A#(12)

See the PUT statement for more examples using GET and PUT.

**Example:**

This example draws a pattern inside a box, then uses GET and PUT to reproduce it twice.

```
10 DIM A%(85)
20 SCREEN 1
30 CLS
40 FOR X = 10 TO 40 STEP 5
50 LINE (X,X)-(X+40,X+40),B
60 NEXT
70 LINE (5,5)-(85,85),B
80 GET (5,5)-(85,85),A%
90 PUT (45,45),A%
100 PUT (85,85),A%
```

## GET and PUT Statements (for COM(n) files)

**Format:** GET *filename*, *nbytes*  
PUT *filename*, *nbytes*

**Purpose:** Permits fixed length I/O for communication.

**Remarks:** *filename* is the number you gave the file when you opened it.

*nbytes* is an integer expression that gives the number of bytes to be transferred into or out of the file buffer. It cannot exceed the value you set with the LEN option in the OPEN "COM" statement.

**Example:** 1000 GET 1, 75

## GOSUB...RETURN Statement

**Format:** GOSUB *line#*...RETURN

**Purpose:** Branches to and returns from a subroutine.

**Remarks:** *line#* is the first line of the subroutine.

Subroutines allow you to key in a group of statements once, yet access them from different parts of a program. The GOSUB statement directs program flow to a subroutine, and sets up the mechanism to return control to the line following the GOSUB statement when the subroutine finishes execution.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by the memory available in the Vectra BASIC stack.

A subroutine's RETURN statement causes Vectra BASIC to branch back to the statement following the most recently executed GOSUB statement. A subroutine may contain more than one RETURN statement when program logic dictates returning from different parts of the subroutine.

Although subroutines may appear anywhere within a program, good programming practice recommends that subroutines be readily distinguishable from the main program. You may precede a subroutine with a STOP, END, or GOTO statement to direct program control around the subroutine. (This prevents program control from inadvertently "falling through" a subroutine.)

**Example:**

```
10 PRINT "MAIN PROGRAM"
20 GOSUB 60
30 PRINT "BACK FROM SUBROUTINE"
40 END
50 REM ***** SUBROUTINE SECTION *****
60 PRINT "SUBROUTINE ";
70 PRINT "IN ";
80 PRINT "PROGRESS"
90 RETURN
RUN
MAIN PROGRAM
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
OK
```

The END statement in line 40 prevents the subroutine from being executed a second time.

---

**GOTO Statement****Format:**

*GOTO line*

**Purpose:**

Branches directly to the specified line number.

**Remarks:**

*line* is the line number of a statement in the program.

When *line* is an executable statement, Vectra BASIC executes that statement and program flow continues from there. When *line* is a nonexecutable statement (such as REM or DATA), execution continues at the first executable statement following *line*.

In Direct Mode, you may use the GOTO statement to reenter a program at a desired point. This can aid debugging.

**Examples:**

The first example demonstrates Indirect Mode.

```
10 READ RADIUS
20 PRINT "RADIUS = "; RADIUS
30 AREA = 3.14 * RADIUS ^2
40 PRINT "AREA = "; AREA
50 GOTO 10
60 DATA 5,7,12
RUN
RADIUS = 5      AREA = 78.5
RADIUS = 7      AREA = 153.86
RADIUS = 12     AREA = 452.16
Out of DATA in 10
OK
```

The next example demonstrates Direct Mode.

```
GOTO 20
RADIUS = 12      AREA = 452.16
Out of DATA in 10
Ok
```

**Note**

You may use the GOTO statement in Direct Mode. However, if you precede this command with any other command that might change the values of variables (such as CLEAR or RESTORE), your results will differ.

**HEX\$ Function**

**Format:**        HEX\$(X)

**Action:**       Returns a string that represents the hexadecimal value of the decimal argument.

Vectra BASIC rounds X to an integer before it evaluates HEX\$(X).

See the OCT\$ function for octal conversions.

**Example:**

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X " DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
32 DECIMAL IS 20 HEXADECIMAL
Ok
```

## IF Statement

### Format 1:

IF *expression* ( , ) THEN (*clause*) (GOTO *line*) (ELSE (*clause*) (*line*))

### Format 2:

IF *expression* GOTO *line* (ELSE (*clause*) (*line*))

### Purpose:

Determines program control based upon the result of the logical expression.

### Remarks:

*expression* is any logical (numeric) expression.

*clause* is either a BASIC statement or a sequence of statements that you separate with colons (:).

*line* is the line number of a statement in the program.

When the result of the *expression* is true (not zero), Vectra BASIC executes the THEN or GOTO clause. Consider this example:

```
10 INPUT I
20 PRINT I
30 IF I THEN GOTO 50
40 STOP
50 PRINT "HI!"
60 END
? 1  Enter
1
HI!
```

When *expression* is false (zero), Vectra BASIC disregards the THEN or GOTO clause and executes the ELSE clause if it is present. Otherwise, execution continues with the next executable statement. Consider this example:

```
10 INPUT I
20 PRINT I
30 IF I THEN GOTO 50
40 STOP
50 PRINT "HI!"
60 END
RUN
? 0  Enter
0
Break in 40
OK
COUNT  Enter
HI!
OK
```

You may follow the reserved word THEN with a line number where program control should branch. The "goto" is implied; IF J=3 THEN 100 is equivalent to IF J=3 THEN GOTO 100 or IF J=3 GOTO 100. All three are correct.

You may also follow the word THEN with one or more Vectra BASIC statements, separated by colons. These statements will be executed only if the expression is true. If the expression is false, program control goes to the next line number in your program, or to the ELSE portion of an IF ... THEN ... ELSE statement, not the next statement.

To include multiple statements in an IF ... THEN ... ELSE program line, use this format:

```
IF expression THEN statement1 ; statement2
ELSE statement3 ; statement4
```

There is no colon between *statement2* and ELSE. If *expression* is true, statements 1 and 2 are executed. If *expression* is false, statements 3 and 4 are executed.

You may place a comma before THEN.

You can only use a line number after the reserved word `GOTO`.

**Nesting of IF Statements:** You may nest `IF...THEN...ELSE` statements to any depth, limited only by the length of the input line (255 characters). For example, the following statement is legal:

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X THEN PRINT  
"LESS THAN" ELSE PRINT "EQUIVALENT"
```

When an `IF` statement contains a different number of `ELSE` and `THEN` clauses, Vectra BASIC pairs each `ELSE` with the closest unmatched `THEN`. In the following example, the single `ELSE` clause pairs with the second `THEN`, not the first.

```
IF A=B THEN IF B=C THEN PRINT "A=C"  
ELSE PRINT "A<>C"
```

When you are conversing with the Vectra BASIC interpreter in Direct Mode and if you follow an `IF...THEN` statement with a line number, the interpreter displays an `Undefined line number` error message unless a line with the specified line number exists in the program in memory.

### Note

When using the `IF` statement to test equality for a value that results from a floating point computation, you should remember that the internal representation of the value is not exact. (This happens because a decimal number is being represented in binary format.) Therefore, you should conduct the test against the range of values over which accuracy may vary. For example, to test a computed variable `A` against the value `1.0`, use:

```
IF ABS (A-1.0) < 1.0E-6 THEN...
```

rather than:

```
IF A=1.0 THEN...
```

The recommended method returns true if the value of `A` is between .999999 and 1.000001 (a relative error of less than  $1.0E-6$ ).

### Examples:

This statement gets record number `I` if `I` is not zero:

```
200 IF I THEN GET #1, I
```

The following program segment tests whether `I` is between 10 and 20. If `I` is within this range, Vectra BASIC calculates a value for `DB` and branches to line 300. If `I` is outside this range, execution continues with line 110:

```
100 IF (I>10) AND (I<20)  
THEN DB=1979 * I : GOTO 300  
110 PRINT "VALUE OUT OF RANGE"  
120 GOTO 100
```

### Note

In the second example above, the `THEN` portion of the statement was entered on 2 lines to increase the readability of the program. The line with the `IF` portion of the statement was terminated with a `[CTRL] [J]`, then a few tab characters were used to indent the `THEN` portion. `[CTRL] [J]` ends a screen line without terminating a logical line.

The next example selects a destination for printed output, depending on the value of a variable (`IDFLAG`). If `IDFLAG` is zero (false), output goes to the line printer; otherwise, output goes to the computer screen:

```
210 IF IDFLAG THEN PRINT A# ELSE LPRINT A#
```

## INKEY\$ Function

**Format:** INKEY\$

**Action:** Returns a character from the keyboard or keyboard buffer in string form.

The returned string can be 0, 1, or 2 characters long.

If no characters are in the keyboard buffer, INKEY\$ returns the null string (length=0).

Most keyboard presses return a one-character string that is the actual character typed. This includes the alphanumeric keys (shifted and unshifted), and the alphabetic keys combined with CTRL.

Certain combinations of keystrokes produce special "extended codes" which return a two-character string.

The cursor control keys when used alone produce extended codes. Other extended codes are produced by a combination of a function key, an alphanumeric key, or cursor control key with ALT, Shift, or CTRL.

The first character of the two character string is always the null character (ASCII 00). For the alphabetic keys, the second character matches the scan code for the key (see Appendix B for a list of scan codes). To obtain a numeric value for the second character returned by INKEY\$, you can use this procedure:

```
10 A$=INKEY$
20 IF LEN(A$)<2 THEN 10
30 I=ASC(RIGHT$(A$,1))
```

The value of I then matches the extended code from this chart:

Extended Code	Key or Key Combination
3	NUL (nul character)
15	Shift & tab
16-25	Alt & Q, W, E, R, T, Y, U, I, O, P
30-38	Alt & A, S, D, F, G, H, J, K, L
44-50	Alt & Z, X, C, V, B, N, M
59-68	Function keys F1-F10 (they must first be disabled as soft keys)
71	Home
72	Cursor Up
73	Pg Up
75	Cursor Left
77	Cursor Right
79	End
80	Cursor Down
81	Pg Dn
82	Ins
83	DEL
84-93	Shift & F1 - F10
94-103	CTRL & F1 - F10
104-113	Alt & F1 - F10
114	CTRL & Prt Sc
115	CTRL & Cursor Left
116	CTRL & Cursor Right
117	CTRL & End
118	CTRL & Pg Dn
119	CTRL & Home
120-131	Alt & 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, =
132	CTRL & Pg Up

INKEY\$ does not echo the typed character on the screen.

The following keystrokes are not intercepted by INKEY\$, but perform their usual functions:

CTRL	Break	Interrupts program execution
CTRL	Alt	Resets the system
CTRL	Alt	Resets the system
CTRL	Alt	Pauses program execution
Shift	Num Lock	Prints the screen contents
	Prt	

#### Examples:

This program loops between lines 20 and 30 until a key is pressed.

```
10 PRINT "PRESS A KEY"
20 AS = INKEY$
30 IF AS = "" THEN GOTO 20
40 PRINT "YOU PRESSED THE "; AS; " KEY"
50 END
```

This example uses the extended codes for the cursor control keys.

```
10 REM Sketcher using INKEY
20 SCREEN 1
30 CLS
40 X=150:Y=100
50 AS=INKEY$
60 IF LEN(AS)<2 GOTO 50
70 IF ASC(RIGHT$(AS,1))=77 THEN X=X+1
80 IF ASC(RIGHT$(AS,1))=75 THEN X=X-1
90 IF ASC(RIGHT$(AS,1))=72 THEN Y=Y-1
100 IF ASC(RIGHT$(AS,1))=80 THEN Y=Y+1
110 PSET (X,Y)
120 GOTO 50
```

## INP Function

#### Format:

INP ( )

#### Action:

Returns the byte read from the input port ( ) max range from 0 to 65535.

#### Note

The input port is a microprocessor port. It does not refer to your computer's datacomm (or peripheral) ports.

INP is the complementary function to the OUT statement.

#### Example:

This example uses an OUT statement to trigger the joystick port, then reads the joystick port with INP. This is equivalent to the STRIG function, but accesses the port directly.

```
10 REM Routine to read the joystick triggers
20 OUT $H201,0
30 A=INP ($H201)
40 IF (A AND $H80) THEN PRINT "A1"
50 IF (A AND $H40) THEN PRINT "A2"
60 IF (A AND $C20) THEN PRINT "B1"
70 IF (A AND $C10) THEN PRINT "B2"
80 GOTO 20
```

## INPUT Statement

- Format:** INPUT { ; } ["prompt" { ; | , } variable { , variable } ...]
- Purpose:** Takes input from the keyboard during program execution. Vectra BASIC accepts the data after you press the **[Enter]** key.
- Remarks:** *prompt* is a string constant that assists the user in entering the proper information.
- variable* is the name of the numeric or string variable that receives the input. The variable may be a simple variable or the element of an array.
- When Vectra BASIC encounters an INPUT statement, it prints a question mark (?) to show that the program is waiting for data. When you include *prompt*, Vectra BASIC displays that string before the question mark. You may then enter the requested data from the keyboard.
- You may use a comma (,) instead of a semicolon after the prompt string to suppress the question mark. For example, the following statement prints the prompt without the trailing question mark:

```
INPUT "ENTER BIRTHDATE ", BDATE
```

When you place a semicolon immediately after the reserved word INPUT, pressing the **[Enter]** key does not echo a carriage return/line feed sequence:

```
10 PRINT "FOR EXAMPLE"
20 INPUT: A$
30 INPUT: B$
RUN
FOR EXAMPLE
? A [Enter]? B [Enter]
Ok
```

As you enter the necessary data, Vectra BASIC assigns the values to the listed variable(s). You must separate a series of items with commas, and the number of items you enter must agree with the number of variables in the list.

- Responding to a prompt with too many or too few items, or the wrong type of value (string instead of numeric, for instance), prints the message *?Redo from start*. Vectra BASIC makes no assignment of values until it receives a completely acceptable response. For example,

```
10 INPUT "ALPHA PLEASE :", A$
20 INPUT "NUMBER ONLY :", B
30 PRINT "*****A$-:", A$
40 PRINT "*****B -:", B
RUN
ALPHA PLEASE :ALFA
NUMBER ONLY :24
*****A$- ALFA
*****B - 24
Ok
RUN
ALPHA PLEASE :BETA
NUMBER ONLY :B
?Redo from start
NUMBER ONLY :48
*****A$- BETA
*****B - 48
Ok
```

When entering string information to an INPUT statement, you may omit surrounding the text with quotation marks.

If the prompt requests a single response, you may press the **[Enter]** key to enter a zero for a numeric item or the null string for a string variable.

**Examples:**

The first example takes the user's input and squares that value.

```
10 INPUT X
20 PRINT X " SQUARED IS " X^2
30 END
RUN
```

? (you type 5 )

```
S SQUARED IS 25
OK
```

The next example calculates the area of a circle.

```
10 PI = 3.14
20 INPUT "WHAT IS THE RADIUS"; R
30 A = PI * R^2
40 PRINT "THE AREA OF THE CIRCLE IS "; A
50 END
RUN
```

WHAT IS THE RADIUS?

(you type 7.4 )

```
THE AREA OF THE CIRCLE IS 171.9464
OK
```

This example finds the average of three numbers.

```
10 INPUT "ENTER THREE VALUES: ", A,B,C
20 AVE = (A+B+C)/3
30 PRINT "THE AVERAGE IS "; AVE
RUN
```

ENTER THREE VALUES:

(you type: 5, 10, 9 )

```
THE AVERAGE IS 8
OK
```

The last example demonstrates how you may screen the input value to determine if it is an appropriate response.

```
10 INPUT "ENTER EMPLOYEE NUMBER"; ID
20 IF ID<25 THEN PRINT " INCORRECT VALUE"
...
RUN
```

(you type 5  to the prompt)

```
ENTER EMPLOYEE NUMBER? 5 INCORRECT VALUE
```

## INPUT # Statement

**Format:**

INPUT # *filenum*, *variable* [, *variable*] . . .

**Purpose:**

Reads data values from a sequential disc file and assigns them to program variables.

**Remarks:**

*filenum* is the number you gave the file when you opened it for input.

*variable* is the name of a numeric or string variable that receives the value read from the file. The variable may be a simple variable or an array element.

The INPUT # statement suppresses printing of the question mark as a prompt character.

Data items in a file should appear exactly as they would if you were typing the information as a response to an INPUT statement.

The items read must match the variable type of each variable.

For numeric values, Vectra BASIC discards any leading spaces, carriage return characters, or line feed characters. The first character that Vectra BASIC encounters that is not a space, carriage return, or line feed character is taken to be the beginning of a number. The number terminates on a space, comma, carriage return, or line feed character.

When Vectra BASIC scans a sequential file for a string value, it also discards any leading spaces, carriage returns, or line feed characters. The first character that it encounters that is not one of these three characters is taken to be the start of a string item. When the first character is a quotation mark ("), the string consists of all characters that occur between the first quotation mark and the second. Thus a quoted string cannot contain embedded quotation marks. When the first character is not a quotation mark, Vectra BASIC considers the string to be unquoted. In this case, the string terminates on a comma, carriage return, or line feed, or after 255 characters have been read.

If Vectra BASIC reaches the end-of-file while reading a numeric or string value, it terminates the item immediately.

**Example:**

```
10 OPEN "I", #1, "BUDG"  
20 INPUT #1, CHCKNUM$, PAYEE$  
30 PRINT CHCKNUM$, PAYEE$  
40 GOTO 20  
  
RUN  
2134      ELECTRIC COMPANY  
2136      GAS BILL
```

### Examples:

The first example lists the contents of a sequential file in Hex:

```
10 OPEN "1",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,1)))
40 GOTO 20
50 PRINT
60 END
```

The next program segment determines program flow based upon a user's response:

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$ = INPUT$(1)
120 IF X$ = "P" THEN 500
130 IF X$ = "S" THEN 700 ELSE 100
. . .
:
```

## INPUT\$ Function

**Format:** INPUT\$(i [, #] *filename*)

**Action:** Returns a string of *i* characters.

*i* is the number of characters to be read from the file.

*filename* is the file number that you used to open a file. Including the *filename* parameter reads the string from that file. If you omit the *filename* parameter, INPUT\$ reads the string from the computer's keyboard. When the keyboard serves as the source of input, INPUT\$ suppresses the echoing of characters to the screen and passes through all characters including control characters. The only exception is **CTRL Break**, which you may use to interrupt the execution of the INPUT\$ function and return control to the Vectra BASIC command level.

The following discussion and program segment pertain to using the INPUT\$ function with COM files:

When reading communication files, you should use the INPUT\$ statement, rather than the INPUT or LINE INPUT statement, since all ASCII characters may be significant in communications.

```
500 REM Read and echo print until a DC1
510 A$ = INPUT$(1,1)
520 IF A$ <> CHR$(17)
    THEN PRINT A$; : GOTO 510
```

For further information, see the OPEN "COM" statement.

## INT Function

**Format:** INT(*x*)

**Action:** Returns the largest integer that is less than or equal to *x*. See the FIX and CINT functions which also return integer results.

**Examples:**

```
PRINT INT(99.89)
99
Ok
PRINT INT(-12.11)
-13
Ok
```

## INSTR Function

**Format:** INSTR(*i*, *x*\$, *y*\$)

**Action:** Searches for the first occurrence of string *y*\$ in *x*\$, and returns the position where the match occurs.

*i* is an offset that determines the starting position for the search. Its value may range from 1 to 255. If the value for *i* is outside this range, an `Illegal function call` occurs.

*x*\$ and *y*\$ may be string variables, string expressions, or string literals.

INSTR returns a value of 0 when:

- *x*\$ is the null string ("")
- the string *y*\$ is not in string *x*\$
- *i* exceeds the number of characters in *x*\$ (`1 > LEN(x$)`).

When *y*\$ is the null string, the function returns 1 (or 1 if you omitted the offset parameter).

**Example:**

In the following example, when the search starts at the string's beginning, the first occurrence of "B" is position 2. However, when an offset parameter skips the first "B", the function returns the position for the next occurrence (that is, position number 6):

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$, Y$); INSTR(3, X$, Y$)
RUN
2 6
Ok
```

## IOCTL\$ Function

**Format:**     IOCTL\$(#) *filename*

**Action:**     Receives control data strings from a device driver.

*filename* is the number you used when you opened the device.

IOCTL\$ is usually used to see if the control strings sent by IOCTL were received properly, or to obtain current status information such as the baud rate of a modem, or the current line length of a printer.

\* The conditions for use which are listed in the IOCTL statement also apply to the use of IOCTL\$.

### Example:

This example continues from the IOCTL statement example; it assumes that the device driver for "MYLPT" echoes the control characters after successfully completing an instruction. In this case, if the control characters are not correctly echoed, the ERROR statement is used to cause an error trap with a user-defined error code.

```
10 OPEN "\DEV\MYLP" FOR OUTPUT AS #1
20 IOCTL #1, "PL66"
30 IF IOCTL(1) <> "PL66" THEN ERROR 215
```

## IOCTL Statement

**Format:**     IOCTL( # ) *filename*, *string*

**Purpose:**     Prints a control character or string to a device driver.

**Remarks:**   *filename* is the number you used when you opened the device.

*string* can be up to 255 characters.

There are 3 conditions for using IOCTL statements:

1. The device driver must be installed.
2. The device driver must state that it processes IOCTL strings.
3. The device must be opened (see the OPEN statement) before IOCTL is used.

The format and the content of the strings is determined by the characteristics of the driver. Most standard MS-DOS drivers don't process IOCTL strings. For more information about using device drivers, see the *HP Vectra MS-DOS User's Guide*.

### Examples:

The following examples assume that a device driver called "MYLPT" is installed which can reset the page length for a printer. The string required by this driver is "PL66" to set the page length to 66 lines:

```
10 OPEN "\DEV\MYLP" FOR OUTPUT AS #1
20 IOCTL #1, "PL66"
```

*string* is the user-assigned value. The string can contain a maximum of fifteen characters. (The editor truncates all excess characters.) Only the first eight characters, however, appear in the label field.

Assigning the null string to a function key disables that key as a function key.

**Examples:** The first example assigns a new label to function key number 5:

```
KEY 5, "NEWLABEL"
```

This statement assigns a new function to key 10. The ASCII character string function adds a return to the end of the line:

```
KEY 10, "PRINT TIMES";CHR$(13)
```

This statement disables function key 7:

```
KEY 7, ""
```

**Format 2:**

```
KEY n, CHR$(qualifier) + CHR$(keycode)
```

**Remarks:**

This form of the KEY statement assigns keystroke combinations to the user-definable trap keys. You can trap combinations of the alphanumeric keys plus Alt, Shift, or Control.

**Note**

See the ON KEY and KEY(*n*) statements for more information on key trapping.

*n* is the user key number. The permissible values are 15 through 20.

*qualifier* and *keycode* uniquely define a key on the keyboard.

*keycode* is a key's SCAN code (not the ASCII value of the character). See Appendix B for a list of scan codes.

## KEY Statement

**Format:**

```
KEY key#, string  
KEY n, CHR$(qualifier) + CHR$(keycode)  
KEY LIST  
KEY ON  
KEY OFF
```

**Purpose:**

Assigns user-defined expressions to the computer's function keys and turns the key display on or off.

**Format 1:**

```
KEY key#, string
```

**Remarks:**

*key#* is a function key number. Each function key has a numeric label from 1 to 10.

**Note**

The eight function keys across the top of the keyboard, F1-F8, duplicate the functions of keys F1-F8 on the left side of the keyboard.

### Format 3:

KEY LIST lists all the function keys showing their key number and definition string.

### Format 4:

KEY ON displays the function keys.

The initial values are:

```
F1 LIST
F2 RUN◀
F3 LOAD"
F4 SAVE"
F5 CONT◀
F6 "LPT1:"
F7 TRON◀
F8 TROFF◀
F9 KEY
F10 SCREEN 0,0,0◀
```

In 80-column width, simultaneously pressing **CTRL** **T** toggles the function key display. In 40-column width, 5 function keys are displayed at once. Simultaneously pressing **CTRL** **T** cycles through the sequence of displaying keys 1-5, keys 6-10, and turning off the display.

### Format 5:

KEY OFF erases the function key labels from the screen display. KEY OFF is frequently used in graphics programs, and must be used whenever you want to print on the 25th screen line.

Merely turning off the function key display with KEY OFF does not deactivate the function keys. You must use a statement such as:

```
FOR I = 10 TO 10: KEY I, "" : NEXT I
```

### Example:

```
10 SCREEN 1 : CLS : KEY OFF
```

*qualifier* is the mask for the latched keys: Right Shift, Left Shift, CTRL, Alt, Num lock and Caps lock. These values must be specified in hex.

Key value for:      Hex Value:

Right Shift Key	&H1
Left Shift Key	&H2
CTRL Key	&H4
Alt Key	&H8
Num lock OFF	&H0
Num lock ON	&H20
Caps lock OFF	&H0
Caps lock ON	&H40

You can add values from the list. To trap **CTRL** **Alt**, you use &H12; to trap **CTRL** **Shift**, use &H7.

When trapping the **Shift** keys, you can use &H1, &H2, or &H3. The key trapping process assumes they are the same.

Setting the qualifier to &H0 traps the keys from the keyboard without the use of the latched keys.

You can trap **CTRL** **Break**, but be sure to leave some way to exit your program, such as a test in the trap routine. Otherwise, you'll have to restart your system.

You can also trap **CTRL** **Alt** **DEL**, but you cannot trap **CTRL** **Alt** **Sys Req**.

This example traps **CTRL** **q**. It does not trap upper case Q, and does not operate as long as the **Caps lock**, **Num lock** or **ScrLock** keys are active.

```
10 KEY 15, CHR$(&H4)+CHR$(&16)
20 KEY (&15) ON
30 ON KEY(&15) GOSUB 3000
...
3000 REM CTRL q trap routine
...
3090 RETURN
```

### Examples:

The KEY(n) ON statement enables the trapping of the specified key by an ON KEY statement. While trapping is enabled, and if a non-zero line number is given in the ON KEY statement, Vectra BASIC checks between each program statement to see if the user has pressed the selected key. If the specified key is pressed, Vectra BASIC executes the ON KEY statement.

The KEY(n) OFF statement disables the event trap. If the user initiates the specified event, Vectra BASIC ignores it.

The KEY(n) STOP also disables the event trap. This option, however, remembers the event if it happens to occur. Then when trapping is enabled with a KEY(n) ON statement, Vectra BASIC executes an ON KEY statement immediately.

**Example:**

The following example shows the sequence for setting a key trap. First, define the key, then define the event-trapping subroutine. Finally, enable trapping by turning the key on.

```
10 KEY 10, ""
20 ON KEY(10) GOSUB 200
...
70 KEY(10) ON
...
200 REM Subroutine for screen
```

'Assign key 10

'Define subroutine

'Enable event trapping

**KEY (n) Statement**

**Format:**

KEY(n) ON  
KEY(n) OFF  
KEY(n) STOP

**Purpose:**

Activates or deactivates trapping of the specified key in a Vectra BASIC program.

**Remarks:**

KEY(n) ON Activates trapping of the given key.  
KEY(n) OFF Deactivates trapping of the given key.  
KEY(n) STOP No trapping occurs but the action is remembered and is executed as soon as the program encounters a KEY(n) ON statement.

n is an appropriate key number:

1 through 10 refer to the function keys

11 through 14 refer to the four cursor-direction keys

15 through 20 refer to six user-defined trap sotkeys

**Note**

Refer to the ON KEY statement for a detailed description on how key trapping works and how you use the KEY(n) statement in conjunction with the ON KEY statement.

If you give the KILL statement for an open file, Vectra BASIC displays the *File already open* error message and cancels the command.

You may use the KILL statement for all types of disc files (program files, random data files, and sequential data files).

Directories can be removed with the RMDIR statement.

**Example:**

The first example deletes PROG.BAS in MIKE's subdirectory under SALES:

```
KILL "SALES\MIKE\PROG.BAS"
```

Although you may use wild cards with the KILL command, you should exercise caution. For example, the next statement deletes CHAP.1, CHAP.2, and so on, but would also delete CHAP.NEW, CHAP.FINAL, and CHAP.BUT if these files existed:

```
KILL "CHAP.*"
```

---

## KILL Command/Statement

**Format:**

```
KILL filename
```

**Purpose:**

Deletes the named file from disc.

**Remarks:**

*filename* is a string expression that contains the filename, and may contain an optional drive designator and path.

If no device designator is included in *filename*, Vectra BASIC uses the current drive. If no path is specified Vectra BASIC searches the current directory for *filename*.

When *filename* is a literal, you must enclose the name in quotation marks.

*filename* must include the extension designator, if one exists.

Although Vectra BASIC provides the .BAS designator as a default file type extension when you save a file, it does not supply a default designator for the KILL statement. For example, if you save a program with the statement:

```
SAVE "MYPRG"
```

Vectra BASIC supplies the extension .BAS for you. However, if you later decide to delete that program, you must supply the file's complete name as in:

```
KILL "MYPRG.BAS"
```

*filename* may contain question marks (?) or asterisks (\*) as wild cards. A question mark matches any single character in the filename. An asterisk matches one or more characters, beginning from that position.

## LEN Function

**Format:**      LEN(*x\$*)

**Action:**      Returns the number of characters in *x\$*. LEN counts all non-printing and blank characters.

**Example:**      In this example, because Vectra BASIC initializes all string variables to the null string, the first PRINT statement prints a value of zero:

```
10 PRINT LEN(X$)
30 X$ = "PORTLAND, OREGON"
40 PRINT LEN(X$)
RUN
0
16
OK
```

## LEFT\$ Function

**Format:**      LEFT\$(*x\$,i*)

**Action:**      Returns a string comprised of the leftmost *i* characters of *x\$*. *i* must be in the range of 0 to 255. When *i* is greater than the number of characters in *x\$*, LEFT\$ returns the entire string. When *i* equals zero, the function returns the null string (a string with zero length).

Also see the MID\$ and RIGHT\$ functions.

**Example:**

```
10 A$ = "BASIC"
20 B$ = LEFT$(A$,2)
30 PRINT B$
RUN
BA
OK
```

## LINE Statement

### Format:

```
LINE ((STEP1(x1,y1))-((STEP1(x2,y2))-(color)
[,B (F)))[,style]
```

### Purpose:

Draws a line or box on the screen.

### Remarks:

*x1,y1* is the starting point and *x2,y2* is the ending point. You may give the coordinates in absolute form or relative form. Specifying relative coordinates requires the STEP option:

```
STEP (xoffset,yoffset)
```

When you use the STEP option for the second coordinate, it is relative to the first coordinate in the statement.

*x1,y1* and *x2,y2* may range from -32768 to 32767. Lines may be drawn to points that are off the screen. The figure is computed as if the off-screen points existed, then the line is clipped at the screen edge.

### Note

This command is only valid on the medium and high resolution graphics screens. The medium resolution screen measures 320 pixels (or dots) by 200 pixels. The high resolution screen measures 640 pixels by 200 pixels.

Note that the statement `LINE -(x2,y2)` is valid. It is the simplest form of the LINE statement. It draws a line from the last point referenced to the point `(x2,y2)`.

## LET Statement

### Format:

```
(LET) variable = expression
```

### Purpose:

Assigns the value of an expression to a variable.

### Remarks:

The reserved word LET is optional as the equal sign suffices when assigning an expression to a variable name.

*variable* is the name of a string or numeric variable that receives the value. It may be a simple variable or the element of an array.

Vectra BASIC evaluates *expression* to determine the value that it assigns to *variable*. The type for *expression* must match the *variable* type (string or numeric), or a `Type mismatch` error occurs.

Vectra BASIC interprets the leftmost equal sign in an expression to be the assignment operator. It treats subsequent equal signs as relational operators. For example, in evaluating the following expression, Vectra BASIC sets the value of A to true (-1) if B is equal to C.

```
A = B = C
```

### Examples:

The first example demonstrates the use of the LET statement:

```
110 LET D = 12
120 LET E = 12^2
130 LET F = 12^4
140 LET SUM = D + E + F
```

The following statements make the identical assignments but omit the word LET:

```
110 D = 12
120 E = 12^2
130 F = 12^4
140 SUM = D + E + F
```

You could use the *style* option to draw a dotted line across the screen by plotting every other point. Since *style* is 16 bits wide, the pattern for a dotted line looks like this:

```
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
```

Every four bits equate to 1010, or an "A" in hexadecimal notation. Thus, the *style* integer for a dotted line is &HAAAA.

### Examples:

The first example draws a line from the most recent point offset by (10,20) to the point (100,200):

```
LINE STEP(10,20)-(100,200)
```

This example draws a line from (10,20) to (110,220) (that is, (10,20) offset by (100,200)):

```
LINE(10,20)-STEP(100,200)
```

This statement draws a box whose opposite corners are at (10,20) and (100,200):

```
LINE(10,20)-(100,200),B
```

This statement draws the same box, then fills it:

```
LINE(10,20)-(100,200),,BF
```

The next example draws an infinite number of lines:

```
10 SCREEN 1
20 CLS
30 LINE -(RND*319, RND*199),RND*3
40 GOTO 30
```

This example uses the *style* parameter to create a blinking box.

```
10 SCREEN 1
20 CLS
30 LINE (70,80)-(250,120),1,B,&H5050
40 LINE (70,80)-(250,120),0,B
50 LINE (70,80)-(250,120),2,B,&H505
60 LINE (70,80)-(250,120),0,B
70 GOTO 30
```

*color* is optional. In medium resolution, it selects a color of the palette chose by the COLOR statement. 0 produces the background color; 1-3 select the three available colors. The default value is 3. Any value between 3 and 255 will produce color 3; any value greater than 255 will produce an illegal function call error.

In high resolution, a *color* of 0 or 2 produces the background color; any other value between 1 and 255 produces the foreground color. A value greater than 255 produces an illegal function call error.

BF is the option to draw a box in the foreground:

B draws a box using the supplied coordinates as opposite corners of the box.

F is the option to fill the box.

*style* is a 16-bit integer mask that Vectra BASIC uses to plot points on the screen. This technique is called **line styling**.

You may use line styling for regular lines and boxes.

Attempting to use *style* for filled boxes (BF) causes a **Syntax error**.

Vectra BASIC uses the current bit in *style* to plot points on the screen. When the bit is 0, Vectra BASIC skips plotting that point. When the bit is 1, Vectra BASIC plots the current point. After each point, Vectra BASIC selects the next bit position for the next point. Once all 16 bits have been read, Vectra BASIC begins again with the first bit position.

### Note

A zero (0) bit skips over the current point on the screen. It does not erase the existing point. Therefore, before using *style*, you may want to draw a background line to guarantee a known background color.

## LINE INPUT Statement

**Format:** LINE INPUT ( ; ) ["prompt";] stringvar

**Purpose:** Enters an entire line (up to 254 characters) to a string variable. No string delimiters are necessary.

**Remarks:** *prompt* is a string literal that Vectra BASIC displays upon the computer screen prior to accepting keyboard input. Including a question mark as part of the prompt requires your putting the question mark character at the end of *prompt*.

Vectra BASIC assigns all characters that occur between the end of the prompt and the end of the line to *stringvar*. (Vectra BASIC determines that a line has ended when you press the **Enter** key, or it has read 254 characters.) However, if Vectra BASIC reads a line feed/carriage return combination, both characters are echoed, but the carriage return is ignored. Vectra BASIC includes the line feed character in *stringvar* and continues reading the input data.

When you immediately follow the reserved words LINE INPUT with a semicolon, pressing the **Enter** key to end the input line does not echo a carriage return/line feed sequence. (That is, the cursor remains on the line where you entered your response.)

You may interrupt the entering of data to a LINE INPUT statement by simultaneously pressing the **CTRL** and **Break** keys. Vectra BASIC returns control to the command level and issues the interpreter's Ok prompt. You may then use the CONT statement to resume execution at the LINE INPUT statement.

**Example:**

```

80 LINE INPUT "CUSTOMER INFORMATION? ";C$
90 PRINT "VERIFY ENTRY: "; C$
...
RUN
CUSTOMER INFORMATION? BEATRICE ISOLDA 9S073
VERIFY ENTRY: BEATRICE ISOLDA 9S073

```

Vectra BASIC Statements, Commands, Functions, and Variables 6-145

The last example illustrates animation:

```

10 Ticktock, simulate a clock with LINE
   and CIRCLE
20 PI = 3.1415926#
30 DEF FNX(R, THETA) = R * COS(THETA*PI/180!) + C
40 DEF FNY(R, THETA) = R * SIN(THETA*PI/180!) + CY
50 RS = 70 : RM = SB : RH = 45
60 CX = 160 : CY = 100 : RC = 90
70 SCREEN 1
80 VIEW
90 LINE (0,0)-(319,199),0,BF
100 CIRCLE (CX,CY),RC,1
110 FOR I = 1 TO 3, 'This example takes time!
120   FOR H1 = 270 TO 630 STEP 30 'Hours
130     LINE (CX,CY)-(FNX(RH,H1),FNY(RH,H1)),1
140     FOR M1 = 270 TO 630 STEP 6 'Minutes
150       LINE (CX,CY)-(FNX(RM,M1),
         FNY(RM,M1)),2
160       FOR S1 = 270 TO 630 STEP 6 'Seconds
170         T$ = TIME$
180         IF T$ = TIME$ THEN 180
190         LINE (CX,CY)-(FNX(RS,S1-6),
         FNY(RS,S1-6)),0
200         LINE (CX,CY)-(FNX(RS,S1),
         FNY(RS,S1)),1
210         LINE (CX,CY)-(FNX(RM,M1),
         FNY(RM,M1)),2
220         LINE (CX,CY)-(FNX(RH,H1),
         FNY(RH,H1)),3
230       NEXT S1
240     NEXT M1
250   NEXT H1
260   NEXT M1
270   LINE (CX,CY)-(FNX(RH,H1),
         FNY(RH,H1)),0
280   NEXT H1
290 NEXT I
300 END

```

6-144 Vectra BASIC Statements, Commands, Functions, and Variables

You will find the `LINE INPUT#` statement especially useful if each line of a data file contains several fields, or if a Vectra BASIC program that was saved in ASCII mode is being read as a data file by another program.

### Example:

```
10 OPEN "0", 1, "LIST"
20 LINE INPUT "BIRTH STATS? ", C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "1", 1, "LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
BIRTH STATS? ELAINA MICHELLE 8 2, 20, SQUEL
ELAINA MICHELLE 8 2, 20, SQUEL
Ok
```

## LINE INPUT# Statement

**Format:** `LINE INPUT# filename, stringvar`

**Purpose:** Reads an entire line (up to 254 characters) from a sequential disc data file and assigns them to the string variable. No string delimiters are required.

**Remarks:** *filename* is the number you gave the file when you opened it for input.

Vectra BASIC assigns the line to *stringvar*. This parameter may be either a string variable or an array element.

The `LINE INPUT#` statement reads all characters in the sequential file up to, but not including, a carriage return character. It then skips over the carriage return (or a carriage return / line feed sequence). The next `LINE INPUT#` statement then reads all the following characters up to the next carriage return character.

### Note

The `LINE INPUT#` statement preserves a line feed / carriage return sequence. For example, if a file contains the following ASCII characters:

```
A C R L F B C R C L F D C R L F E L F C R F C R L F
```

then the following program:

```
10 OPEN "1", #1, "FILE"
20 FOR J = 1 TO 4
30 LINE INPUT #1, C$
40 NEXT J
```

returns the following values to `C$`:

1st iteration: A

2nd iteration: B

3rd iteration: C L F D

4th iteration: E L F C R F

You may use a period (.) for either line number to indicate the current line. For example, you could list all the lines from the beginning of the program to the current line with this command:

```
LIST -.
```

*dev* is a string expression that returns a predefined string. Permissible values are `SCRN:`, `LPT1:`, `LPT2:` and `LPT3:`. (The colon is optional for the printer names, but must be included with `SCRN:`.)

When *dev* is a literal, you must enclose the device name in quotation marks.

- filename* is a string expression that "names" a file for future references. *filename* may contain an optional drive designator and path.

If no device designator is included in *filename*, Vectra BASIC uses the current drive. If no path is specified, Vectra BASIC saves the file in the current directory.

Vectra BASIC supplies the filename extension `.BAS` if no extension is specified.

If *filename* is a literal, you must include the name in quotation marks.

When a file already exists on the directory with *filename*, Vectra BASIC overwrites it. No warning is given.\*

The file is saved in ASCII format rather than compressed binary form (as with the `SAVE` command). The `MERGE` command requires ASCII files, and ASCII program files may also be read as data files.

When you omit the *dev* or *filename* parameter with the `LIST` statement, Vectra BASIC lists the lines to the computer's screen. When you omit the *dev* parameter with `LLIST`, the lines are listed to the default printer (`LPT1:`, unless you have changed it.)

---

## LIST and LLIST Command

### Format:

```
LIST (first.line) [- (last.line) ] [ dev | filename ]  
LLIST (first.line) [- (last.line) ] [ dev ]
```

### Purpose:

Lists all or part of the program currently in computer memory to the screen; or, if `LLIST` is used, to a line printer.

### Remarks:

*first.line* is the first line to be listed while *last.line* is the last line to be listed. Both must be valid line numbers within the range of 0 to 65529.

When you omit both line number parameters, the listing begins with the first line of the program and goes to the end of the program.

Specifying `first.line` prints only that line.

Specifying `first.line` - prints that line through the end of the program.

Specifying `-last.line` prints all lines from the beginning of the program through the given line.

Specifying `first.line-last.line` prints all the lines within that range.

## LOAD Command

### Format:

LOAD *filename* [ , R ]

### Purpose:

Loads a Vectra BASIC program file from disc into your computer's memory.

### Remarks:

*filename* is a string expression that contains the filename that you used to name the program when you saved it. *filename* may contain an optional drive designator and path.

If no device designator is included in *filename*, Vectra BASIC uses the current drive. If no path is specified, Vectra BASIC searches the current directory for *filename*.

Vectra BASIC will supply the filename extension `.BAS` if no extension is specified.

If filename is a literal, you must enclose the name in quotation marks.

Before it loads the named program, Vectra BASIC closes all open files and deletes all variables and program lines that currently reside in BASIC memory. The `R` option, which runs the program after it is loaded, leaves data files open (current program lines and variables are still deleted.) Thus, you may use the `LOAD` command with the `R` option to chain together several programs (or segments of the same program). You pass information between the programs through shared data files.

In any event, you can stop the listing by pressing `CTRL Break`.

Vectra BASIC always returns control to the command level after a `LIST` or `LLIST` command executes.

### Note

The `LLIST` command assumes a printer line width of 80 characters.

### Examples:

The first example lists the program currently stored in your computer's memory:

```
LIST
```

The next statement lists only line 500:

```
LIST 500
```

The next example lists all program lines from line 50 through the end of the program:

```
LIST 50 -
```

The next statement lists all program lines from the program's first line through line 50:

```
LIST -50
```

This example lists lines 50 through 80, inclusively.

```
LIST 50-80
```

This command saves lines 1000 to 3000 as an ASCII format file called "Ovrlay1".

```
LIST 1000-3000,"OVRLAY1"
```

### Note

The BASIC compiler offers no support for this command.

## LOC Function

**Format:**        `LOC(filename)`

**Action:**        With random-access files, LOC returns the record number of the last record referenced in a GET or PUT statement.

With sequential files, LOC returns the number of sectors (that is, 128 byte blocks) read from or written to the file since it was opened.

When you open a file for sequential input, Vectra BASIC reads the first sector of the file. Therefore, LOC always returns a "1" even before any input from the file occurs.

*filename* is the number you gave the file when you opened it.

**Example:**        `200 IF LOC(1) > 50 THEN STOP`

### Examples:

The first example loads and runs the program TESTRUN:

`LOAD "TESTRUN",R`

The next example loads the program MYPROG from the disc in drive C but does not run the program:

`LOAD "C:MYPROG"`

### Note

The BASIC compiler offers no support for this command.

## LOCATE Statement

<b>Format:</b>	<code>LOCATE (row) (,col) (,cursor) (,start) (,end)</code>
<b>Purpose:</b>	Moves the cursor to the specified location on the screen.
<b>Remarks:</b>	<p><i>row</i> is the desired line number. It can range from 1 to 24 with the function keys displayed. If the function keys are turned off, <i>row</i> can range from 1 to 25.</p> <p><i>col</i> is the desired column number. When the <code>WIDTH</code> is 40, <i>col</i> can range from 1 to 40; when the <code>WIDTH</code> is 80, <i>col</i> can range from 1 to 80.</p> <p>Values outside the boundaries for <i>row</i> or <i>col</i> cause an illegal function call error message. In this case, the previous settings are retained.</p> <p><i>cursor</i> is a Boolean value that tells whether the monitor should display the cursor while Vectra BASIC is executing Direct Mode statements or a program. By default, whenever a program is run or a statement is executed, <i>cursor</i> is set to 0. If your program sets <i>cursor</i> to 1, the monitor displays the cursor while the program is running.</p> <p>The cursor is always displayed during <code>INPUT</code> statements; setting <i>cursor</i> to 0 does not inhibit the display of the cursor during <code>INPUT</code> statements.</p> <p>The following parameters are valid only in text mode:</p> <p><i>start</i> indicates the line within the character height where the cursor starts.</p> <p><i>stop</i> indicates the line within the character height where the cursor stops.</p>

## LOC Function (for COM files)

<b>Format:</b>	<code>LOC (filename)</code>
<b>Action:</b>	<p>Returns the number of characters in the input queue that are ready to be read.</p> <p>The default size for the input buffer is 256 bytes. (You may change this value with the <code>/C</code>: switch in the Vectra BASIC command line.) If more than 255 characters exist in the queue, the function still returns 255.</p> <p>When fewer than 255 characters remain in the queue, Vectra BASIC returns the actual number of characters that can be read from the device.</p> <p>For further information, see the <code>OPEN</code> "COM statement.</p>

### Example:

```
500 REM Read all characters in buffer
510 A$ = INPUT$( LOC(1), #1)
```

Once you have moved the cursor to a specific position, subsequent **PRINT** statements begin printing from that location.

Consider this example:

```
10 CLS
20 LOCATE 5,15
30 PRINT "HI"
40 LOCATE 10,25
50 PRINT "THERE"
60 LOCATE 12,75
70 PRINT "LINE OVERFLOWED"
```

As Vectra BASIC cannot fit the last message on the specified line, it begins printing in the first column position of line 13.

Also see the descriptions for the **CURLIN**, **POS**, and **LPDS** functions.

### Examples:

The first example moves the cursor to the upper-left corner of the screen:

```
100 LOCATE 1,1
```

The following statement turns on the cursor at its current position:

```
200 LOCATE ,,1
```

This example uses the **ON TIMER** trap which directs program control to line 2000. This line prints the current system time in row 25.

```
10 KEY OFF
20 ON TIMER(10) GOSUB 2000
30 TIMER ON
...
2000 LOCATE 25,1:PRINT TIME$
2010 RETURN
```

*start* and *stop* control the cursor's height and its location with respect to the characters.

*start* indicates the highest scan line of the cursor (the top of the character), and the *stop* indicates the lowest scan line.

*start* and *stop* can both range from 0 to 31. With a color display adapter, scan lines 0 to 7 are displayed; with a monochrome display adapter, 0 through 13 are displayed.

If *stop* is less than *start*, for example **LOCATE 1,1,1,5,2**, the cursor wraps around the character position. With a color display adapter, the cursor will be composed of scan lines 6,7,0,1, and 2; with a monochrome adapter, it will be composed of lines 6-13 and 0-2.

### Note

Some video adapters, including the HP Multimode card, do not display "wrapped" cursors, where *start* is greater than *stop*. On these adapters, the cursor will become invisible.

When you omit a parameter (except *stop*), the parameter's present status remains in effect. For example, omitting the final 3 parameters does not affect the appearance of the cursor. Omitting the first two parameters (as in **LOCATE , ,1,0,7**) does not change the cursor's position.

If the *stop* parameter is omitted, it assumes the value of the *start* parameter.

### Note

You may only use a row value of 25 if you have turned off the function key display with a **KEY OFF** statement or a **Control-T**. Row 25 does not scroll. The final example shows one possible use for this feature.

## LOF Function (for COM files)

**Format:**      LOF (*filename*)

**Action:**      Returns the amount of free space in the input queue.

You can use this function to check when the input queue is getting full. That is, the function returns a value equal to the size of the communications buffer minus the number of bytes already allocated to the file.

The default size for the communications buffer is 256 bytes, but you may change this value with the /C: switch on the Vectra BASIC command line.

For further information, see the OPEN \*COM statement.

### Example:

The following example depends upon a user-defined error code (see the ERROR statement for details):

```
500 REM Check for buffer overflow
510 IF LOF(1) < 2 THEN ERROR 245
```

## LOF Function

**Format:**      LOF (*filename*)

**Action:**      Returns the length of the file in bytes.

*filename* is the number you gave the file when you opened it.

### Example:

In this example, the variables REC and RECSIZE contain the record number and record length. The calculation determines whether the specified record is beyond the end-of-file.

```
90 IF REC * RECSIZE > LOF(1)
   THEN PRINT "INVALID ENTRY"
```

## LPRINT and LPRINT USING Statements

- Format:** LPRINT *list of expressions* 1  
LPRINT USING *stringexp* ; *list of expressions*
- Purpose:** Prints data to a line printer.
- Remarks:** These statements are identical to PRINT and PRINT USING, except output goes to a line printer. For details of operation, see the PRINT and PRINT USING statements in this chapter.
- ♦ LPRINT assumes that the printer has a line width of 80 characters. This may be reset with the WIDTH statement.
- Example:** LPRINT "THIS IS A TEST"

## LOG Function

- Format:** LOG(*x*)
- Action:** Returns the natural logarithm of *x*.  
*x* must be a positive number.
- Example:** PRINT LOG(45/7)  
1.860752  
OK

## LPOS Function

- Format:** LPOS(*n*)
- Action:** Returns the current position of the line printer print head within the line printer buffer. This may differ from the physical position of the print head.
- n* is an index for the printer: LPOS(1) tests LPT1:, LPOS(2) tests LPT2:, and LPOS(3) tests LPT3:. LPOS(0) also tests LPT1:. All other values for *n* produce an illegal function call error.
- Example:** 100 IF LPOS(1) > 132 THEN LPRINT CHR\$(13)

**Example:**

```
10 OPEN "R", #1, "FILE", 24
20 FIELD #1, 4 AS AMT$, 20 AS DESC$
30 INPUT "PRODUCT CODE"; CODE$
40 INPUT "PRICE"; PRICE
50 INPUT "DESCRIPTION"; DSCRPN$
60 LSET AMT$ = MK$(PRICE)
70 LSET DESC$ = DSCRPN$
80 PUT #1, CODE$
90 GOTO 30
```

---

**LSET and RSET Statements****Format:**

```
LSET stringvar = stringexp
RSET stringvar = stringexp
```

**Purpose:**

Moves data from memory to a random file buffer (in preparation for a PUT (for files) statement).

**Remarks:**

*stringvar* is the name of a variable that you defined in a FIELD statement.

*stringexp* identifies the information that is to be placed into the field named by *stringvar*.

When *stringexp* requires fewer bytes than were allocated to *stringvar*, LSET left-justifies the string in the field, while RSET right-justifies the string. (Spaces pad the extra positions.) When a string is too long for the field, the excess characters are dropped from the right.

You must use the MKI\$, MKS\$, or MKD\$ function to convert numeric values to strings before you move them into the random file buffer with a LSET or RSET statement.

**Note**

You may also use LSET and RSET to left-justify or right-justify a string in a given field. For example, the following program lines right-justify the string NOTE\$ in a 20-character field:

```
110 A$ = SPACE$(20)
120 RSET A$ = NOTE$
```

You will find these statements helpful when formatting output to a printer.

### Example:

This example shows how the merge command replaces or adds lines to the program currently in memory based upon each program's line numbers.

(Merge File = FILE2)

```
15 REM THIS FILE CHANGES THE LOOP CONTENTS
30 COUNT = COUNT + 1
40 PRINT COUNT

LOAD "FILE1" [Enter]
LIST [Enter]
10 REM THIS FILE IS THE RESIDENT FILE
20 FOR I = 1 TO 10
30 PRINT "HELLO";
50 NEXT I
60 PRINT "DONE"
OK
MERGE "FILE2" [Enter]
OK
LIST [Enter]
10 REM THIS FILE IS THE RESIDENT FILE
15 REM THIS FILE CHANGES THE LOOP CONTENTS
20 FOR I = 1 TO 10
30 COUNT = COUNT + 1
40 PRINT COUNT
50 NEXT I
60 PRINT "DONE"
OK
```

### Note

The BASIC compiler offers no support for this command.

## MERGE Command

### Format:

MERGE *filename*

### Purpose:

Incorporates statements contained in the specified file into the program that currently resides in your computer's memory.

### Remarks:

*filename* is a string expression that contains the filename, and may contain an optional drive designator and path.

If no device designator is included in *filename*, Vectra BASIC uses the current drive. If not path is specified, Vectra BASIC searches the current directory for *filename*.

Vectra BASIC will supply the filename extension .BAS if no extension is specified.

If filename is a literal, you must enclose the name in quotation marks.

You must use ASCII format when you save the file that you want to merge. (That is, you must specify the A option when you give the SAVE command or use LIST to write the program to disc.) When Vectra BASIC detects another format, it displays a Bad file mode error message. If this happens, Vectra BASIC cancels the MERGE command and the program in memory remains unchanged.

You may view the MERGE command as "inserting" the lines from the program on disc into the program in memory. When both programs have identical line numbers, the lines from the disc file replace the corresponding lines in memory.

## MID\$ Statement

**Format:**

MID\$(x\$,i[,j]) = y\$

**Purpose:**

Replaces a portion of one string with another string.

**Remarks:**

x\$ is a string variable or an array element. GW BASIC replaces the designated characters of this string.

i is an integer expression that may range from 1 to 255. It marks the starting position in x\$ where the replacement begins.

j is an integer expression that may range from 0 to 255. It gives the number of characters from y\$ that Vectra BASIC uses in the replacement. When you omit this parameter, Vectra BASIC uses the entire y\$ string.

**Note**

The length of x\$ is fixed. Therefore, if x\$ is five characters long and y\$ is ten characters long, Vectra BASIC only replaces x\$ with the first five characters of y\$.

**Example:**

```
10 A$ = "KANSAS CITY, MO"  
20 MID$(A$,14) = "KS"  
30 PRINT A$  
RUN  
KANSAS CITY, KS  
OK
```

## MID\$ Function

**Format:**

MID\$(x\$,i[,j])

**Action:**

Returns a string of length j characters that begins with the *i*th character in string x\$.

x\$ is any string expression.

i is an integer expression that may range between 1 to 255. j is an integer expression that may range between 0 and 255. Numbers outside these ranges produce an illegal function call.

When you omit the length parameter j, or if fewer than j characters exist to the right of the *i*th character, MID\$ returns all the characters beginning with the *i*th character.

When the starting point i exceeds the length of x\$, MID\$ returns the null string.

Also see the LEFT\$ and RIGHT\$ functions.

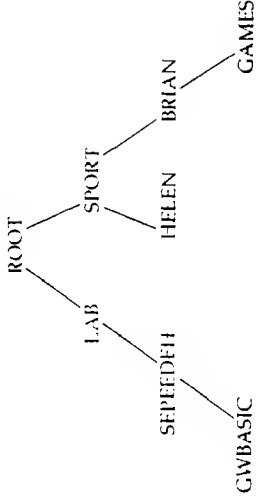
**Example:**

```
10 A$ = "GOOD "  
20 B$ = "MORNING EVENING AFTERNOON"  
30 PRINT A$; MID$(B$,9,7)  
RUN  
GOOD EVENING  
OK
```

Finally, create the subdirectory GAMES under BRIAN:

```
MKDIR "BRIAN\GAMES"
```

By following these steps, you have created this directory tree:



## MKDIR Statement

**Format:**

```
MKDIR path
```

**Purpose:**

Creates a directory on the specified disc.

**Remarks:**

*path* is a string expression (not exceeding 63 characters) that identifies the newly created directory.

**Examples:**

This example assumes that the current directory is the ROOT directory.

Create two subdirectories under the ROOT directory with these statements:

```
MKDIR "LAB"
MKDIR "SPORT"
```

Create the subdirectory SEPEDEH under LAB, then create the subdirectory Vectra BASIC under SEPEDEH:

```
MKDIR "LAB\SEPEDEH"
MKDIR "LAB\SEPEDEH\GW BASIC"
```

Make SPORT the current directory, then create two subdirectories called HELEN and BRIAN:

```
CHDIR "SPORT"
MKDIR "HELEN"
MKDIR "BRIAN"
```

## NAME Statement

- Format:** NAME *oldname* AS *newname*
- Purpose:** Changes the name of a file to the newly given name, and/or moves a file between directories on the same disc.
- Remarks:**
- oldname* is a string expression for the name you gave the file when you opened it or saved it.
  - newname* is also a string expression that conforms to the rules for a valid filename.
  - oldname* and *newname* may contain an optional drive designator and path in addition to a filename. When *oldname* and *newname* contain a drive designator, the drive must be the same.
  - Attempting to rename a file on a different disc produces a **Rename across disks error**.
  - If no drive designator is included in *oldname* or *newname* Vectra BASIC uses the current drive. If no path is specified, Vectra BASIC searches the current directory for filename.
  - If the file is a .BAS file, you must include the file type .BAS in the file's name. Vectra BASIC does not supply .BAS as a default type for you.
  - If *oldname* or *newname* is a literal, you must enclose the name in quotation marks.
  - A file must exist with *oldname*. Similarly, no file can exist with *newname*. When Vectra BASIC fails to find *oldname*, it gives a **File not found error**. Likewise, if Vectra BASIC finds that a file already exists with *newname*, it displays the message **File already exists**.
  - oldname* must be closed before the renaming operation.

## MKIS, MKSS\$, MKD\$, MKDS\$ Functions

- Format:**
- MKIS** (*integer-expression*)
  - MKSS\$** (*single-precision-expression*)
  - MKD\$** (*double-precision-expression*)
- Action:** Converts numeric values to string values.
- Random-access disc files store numeric values as strings. Therefore, when you place values in a random disc file by using the LSET or RSET statement, you must first convert the numbers to strings.
- MKIS** converts an integer to a 2-byte string.
- MKSS\$** converts a single-precision number to a 4-byte string.
- MKD\$** converts a double-precision number to a 8-byte string.
- See also CVI, CVS, and CVD for the complementary operations.
- Example:**
- ```
100 AMT = (K+T)
110 FIELD #1, @ AS D$, 20 AS N$
120 LSET D$ = MKS$(AMT)
130 LSET N$ = A$
140 PUT #1
```
- Note** If you plan to compile your program, see the BASIC compiler manual for differences between the compiled and interpretive versions of these functions.

## NEW Command

### Format:

NEW

### Purpose:

Deletes the program that currently resides in computer memory and clears all variables.

### Remarks:

You use the NEW command in Direct Mode to clear extraneous information from your computer's memory before you enter a new program.

- NEW closes all open files and turns tracing off. Control returns to the command level after this statement executes.

### Example:

Ok  
NEW  
Ok

### Note

The BASIC compiler offers no support for this command.

If you are moving a file to the root directory, there must be room for an additional file name in \. Otherwise, a Too many files error occurs. If you already have several other files open, trying to rename a file may result in the Too many files error because there are not enough file handles available.

### Note

You cannot rename directories with the NAME statement.

### Examples:

The following statement changes the name of the file ACCTS to LEDGER on drive C. After the NAME statement executes, the file still resides on the same area of disc space on the same disc, but with the new name.

```
NAME "C:ACCTS" AS "C:LEDGER"
```

The following example renames a file across directories. The current directory must have a subdirectory called HELEN. This statement moves the file from the current directory to the subdirectory HELEN.

```
NAME "ACCTS" AS "HELEN\ACCTS"
```

This example moves the file to the subdirectory HELEN, and changes the name of the file:

```
NAME "BOOKS" AS "HELEN\LEDGER"
```

## ON COM(n) Statement

**Format:** ON COM(*n*) GOSUB *line*

**Purpose:** Gives the line number of a subroutine that Vectra BASIC is to perform when activity occurs on the specified communications channel.

**Remarks:** *n* is the number of the communications channel. Permissible values for *n* are 1 or 2.

*line* is the first line number of the event trapping routine.

- Setting *line* to zero disables the communications event trap.

The ON COM statement does not start event trapping; it merely specifies the line number of the subroutine. A COM(*n*) ON statement must be used to start event trapping.

**Note** See the COM(*n*) statement for further details.

When event trapping is enabled, and if *line* is not equal to zero, Vectra BASIC checks between statements to see if any activity has occurred on the specified communications channel. When activity occurs, Vectra BASIC transfers control to the subroutine.

When a program executes a COM(*n*) OFF statement, Vectra BASIC ignores any activity on the communications channel.

When a program executes a COM(*n*) STOP statement, Vectra BASIC "remembers" any activity on the channel and executes the GOSUB statement as soon as a COM(*n*) ON statement executes.

## OCT\$ Function

**Format:** OCT\$(*X*)

**Action:** Returns a string that represents the octal value of the decimal argument. Vectra BASIC rounds *X* to an integer before it evaluates OCT\$(*X*).

See the HEX\$ function for hexadecimal conversion.

**Example:**

```
PRINT OCT$(24)
30
OK
```

## ON ERROR GOTO Statement

**Format:** ON ERROR GOTO *line*

**Purpose:** Enables error trapping and specifies the first line of the error-handling subroutine.

**Remarks:** *line* is the line number of the first line of an error-handling routine. If the line number does not exist, an **Undefined line number** error occurs.

Once you have enabled error trapping, Vectra BASIC sends program control to the specified line number whenever it detects an error. (This also includes Direct Mode errors, such as syntax errors.)

You may use the **RESUME** statement to leave an error-handling routine.

You may disable error trapping by executing an **ON ERROR GOTO 0** statement. Any subsequent errors print an error message and halt execution. Within an error-trapping subroutine, the **ON ERROR GOTO 0** statement halts Vectra BASIC and prints the error message for the error that triggered the trap. We recommend that all error-trapping subroutines execute an **ON ERROR GOTO 0** statement if an error is encountered for which no recovery action exists.

### Note

If an error occurs during execution of an error-handling subroutine, Vectra BASIC prints an error message and halts execution. Further error trapping does not occur within an error-handling subroutine.

Event trapping only happens when Vectra BASIC is running a program. When an error trap (resulting from an **ON ERROR** statement) occurs, all trapping is automatically disabled. This includes all **ERROR**, **COM(n)**, and **KEY(n)** statements.

After a trap occurs, an automatic **COM(n)** **STOP** takes place. Thus, recursive traps never occur. A **RETURN** statement from the trap routine automatically does a **COM(n)** **ON** unless a **COM(n)** **OFF** statement is executed inside the trap routine.

You can use a **RETURN line#** statement in the trapping routine to return to a specific line number. However, you must exercise caution when using the **RETURN** statement in this manner. For example, any other **GOSUBs**, **WHILEs**, or **FORs** that were active when the trap occurred will remain active.

### Example:

```
150 ON COM(1) GOSUB 500
160 COM(1) ON
...
500 REM Incoming characters
...
590 RETURN
```

## ON. . .GOSUB Statement

**Format:** ON *result* GOSUB *line1*, *line2* ...

**Purpose:** Branches to one of several, specified subroutines depending upon which value is returned from the governing expression.

**Remarks:** *result* is a numeric expression which must return a value between 0 and 255. (Vectra BASIC rounds the expression to an integer value when necessary.) Any value outside this range causes an **Illegal function call** error.

*line1* is the beginning line number for a subroutine.

The value of *result* determines which subroutine will be used. If the value of *result* is 1, program control branches to the first line number in the list; if *result* is 2, control branches to the second number, and so forth.

In the ON. . .GOSUB statement, each line number in the list must be the first line number of a subroutine.

When the value of *result* is zero or greater than the number of items in the list, Vectra BASIC continues with the next executable statement.

### Example:

```
20 INPUT "ENTER TRIG FUNCTION"; A$
30 IF A$ = "SIN" THEN F = 1 : GOTO 70
40 IF A$ = "COS" THEN F = 2 : GOTO 70
50 IF A$ = "TAN" THEN F = 3 : GOTO 70
60 PRINT "ILLEGAL ENTRY TRY AGAIN" : GOTO 20
70 FOR K = 0 TO 360 STEP 10
80   PRINT K;
90   A = K/180*3.14159
100  ON F GOSUB 1000, 2000, 3000
110  NEXT K
120 STOP
999 REM SUBROUTINE SECTION
1000 SIN(A) : RETURN
2000 PRINT COS(A) : RETURN
3000 PRINT TAN(A) : RETURN
```

Vectra BASIC Statements, Commands, Functions, and Variables 6-170

### Examples:

The following program segments illustrate the effects of the ON ERROR and RESUME statements:

```
5 REM Example without RESUME
10 ON ERROR GOTO 40
20 Y = 9 : Z = 0
30 X = Y/Z 'Division by zero
40 PRINT "ERROR ENCOUNTERED IN LINE 40"; ERL
50 END
RUN
ERROR ENCOUNTERED IN LINE 30
Ok

8 REM With RESUME, execution continues
9 REM on line where the error occurred
10 ON ERROR GOTO 60
20 Y = 9 : Z = 0
30 X = Y/Z
40 PRINT "CONTINUE PROGRAM"
50 GOTO 90
60 PRINT "ERROR ENCOUNTERED IN LINE 40"; ERL
70 Z = 5
80 RESUME
90 PRINT "END"
100 END
RUN
ERROR ENCOUNTERED IN LINE 30
CONTINUE PROGRAM
END
Ok
```

While in Direct Mode, all errors default to the ON ERROR statement:

```
30 PRINT "THIS SYNTAX IS NO GOOD!!"
ON ERROR GOTO 30
Ok
PRINT "ERROR"
THIS SYNTAX IS NO GOOD!!
No RESUME in 30
Ok
```

### Note

If you plan to compile a program that uses the ON ERROR GOTO statement, please refer to the BASIC compiler manual. Also, set the compiler switches properly so your event trapping routine works correctly.

## ON KEY Statement

**Format:** ON KEY(*n*) GOSUB *line*

**Purpose:** Gives the line number where program control goes when the user presses the specified key.

**Remarks:** *n* is an appropriate key number:

- 1 through 10 refer to the function keys
- 11 through 14 refer to the four cursor-direction keys
- 15 through 20 refer to six user-defined trap softkeys

### Note

See the KEY statement for further details.

*line* is the beginning line number of the trapping routine. Setting *line* to zero stops trapping of that key.

A KEY(*n*) ON statement must be active to enable key trapping by the ON KEY(*n*) statement.

When a KEY(*n*) OFF statement executes, key trapping is not performed, and key events are not remembered.

A KEY(*n*) STOP statement suspends ON KEY(*n*) trapping. The GOSUB is not executed immediately, but the event is remembered. When a KEY(*n*) ON statement is executed, the GOSUB is performed.

## ON...GOTO Statement

**Format:** ON *result* GOTO *line 1*, *line 2*, ...

**Purpose:** Branches to one of several specified line numbers, depending upon which value Vectra BASIC returns when it evaluates the controlling expression.

**Remarks:** *result* is a numeric expression which must return a value between 0 and 255. (Vectra BASIC rounds the expression to an integer value when necessary.) Any value outside this range causes an illegal function call error.

*line* is the line number where you want program control to go.

The value of *result* determines which subroutine will be used. If the value of *result* is 1, program control branches to the first line number in the list; if *result* is 2, control branches to the second number, and so forth.

When the value of *result* is zero or greater than the number of items in the list, Vectra BASIC continues with the next executable statement.

### Example:

```
10 REM Simple selection program
20 INPUT "ENTER SELECTION FROM MENU"; K
30 ON K GOTO 50, 70, 90
40 PRINT "INVALID SELECTION" : GOTO 20
50 PRINT "YOU CHOSE SELECTION NUMBER 1"
60 GOTO 20
70 PRINT "YOU CHOSE SELECTION NUMBER 2"
80 GOTO 20
90 PRINT "YOU CHOSE 3 TO END THIS PROGRAM"
100 END
RUN
ENTER SELECTION FROM MENU? 0 
INVALID SELECTION
ENTER SELECTION FROM MENU? 2 
YOU CHOSE SELECTION NUMBER 2
ENTER SELECTION FROM MENU? 3 
YOU CHOSE 3 TO END THIS PROGRAM
OK
```

Event trapping only happens when Vectra BASIC is running a program. When an error trap (resulting from an **ON ERROR** statement) occurs, all trapping is automatically disabled. This includes all **ERROR**, **COM(n)**, **KEY(n)**, **PEM**, **PLAY**, and **STRIG** statements.

After a trap occurs, an automatic **KEY(n)** **STOP** takes place. Thus, recursive key traps never occur. A **RETURN** statement from the trap routine automatically does a **KEY(n)** **ON** unless a **KEY(n)** **OFF** statement executed inside the trap routine.

When a key is trapped, that occurrence of the key is destroyed. Therefore, you cannot subsequently use the **INPUT** or **INKEY\$** statements to find which key caused the trap. If you wish to assign different functions to particular keys, you must set up separate subroutines for each key, rather than assigning the various functions within a single subroutine.

You can use a **RETURN line#** statement in the trapping routine to return to a specific line number. However, you must exercise caution when using the **RETURN** statement in this manner. For example, any other **GOSUBs**, **WHILEs**, or **FORs** that were active when the trap occurred will remain active.

You may define user softkeys with this statement:

```
KEY n, CHR$(qualifier) + CHR$(keycode)
```

See the discussion of the **KEY** statement.

The following rules govern the sequence in which Vectra BASIC traps keys:

- Vectra BASIC processes Shift/Print combination first. Defining the **[PrtScr]** key as a user-defined key trap does not prevent characters from being sent to the Line Printer if the user presses this keystroke combination.
- Vectra BASIC next examines the function keys and the four cursor-direction (arrow) keys. Defining any of these keys to be a user-defined key trap has no effect since Vectra BASIC considers them to be predefined.
- Finally, Vectra BASIC examines the user-defined keys.

Trapped keys are not passed on. (That is, Vectra BASIC does not receive this input.) This applies to all keys, including Control-[Break]. Thus, it is possible for you to prevent application users from accidentally "breaking" out of a program.

### Caution

Be careful when trapping Control-[Break]. If a program is trapping Control-[Break] and no other exit exists, the user will have to reset the machine (with possible loss of data) to stop the program.

### Example:

```
10 KEY 15, CHR$(0) + CHR$(4H10)
20 ON KEY (15) GOSUB 1000
30 KEY (15) ON
...
1000 REM Subroutine for trapped keys
1010 PRINT "TRAPPED LOWER CASE q"
1020 RETURN
```

## ON PEN Statement

**Format:** ON PEN GOSUB *line*

**Purpose:** Gives the line number where program control goes when the lightpen is used.

**Remarks:** *line* is the first line number of the trapping routine. A *line* of zero disables the pen event trap.

A PEN ON statement must be active to enable pen trapping by ON PEN. If trapping is enabled, and the line number of the ON PEN statement is not zero, Vectra BASIC checks between statements to see if the lightpen has been activated. If the lightpen has been used between the statements, a GOSUB is performed to the specified line.

A PEN OFF statement disables pen event trapping. The GOSUB is not executed, and the event is not remembered.

A PEN STOP statement suspends pen event trapping. The GOSUB is not performed, but it is remembered and it will be performed as soon as a PEN ON statement is executed.

### Note

See the PEN statement and the PEN function for more information about using the lightpen.

When an event trap is performed, (that is, when the GOSUB is executed) an automatic PEN STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine automatically performs a PEN ON unless an explicit PEN OFF was performed inside the subroutine.

You can use a RETURN *line#* statement in the trapping routine to return to a specific line number. However, you must exercise caution when using the RETURN statement in this manner. For example, any other GOSUBs, WHILEs, or FORs that were active when the trap occurred will remain active.

Event trapping only happens when Vectra BASIC is running a program. When an error trap (resulting from an ON ERROR statement occurs, all trapping is automatically disabled.

This includes all ERROR, COM(n), KEY(n), PEN, PLAY, STRING(n) and TIMER statements.

When pen trapping is enabled, Vectra BASIC checks before each statement to see if the lightpen has been activated. If portions of your program do not use the lightpen, you can speed execution by turning using the PEN OFF statement before these sections.

### Note

When pen event trapping takes place the action that springs the trap is NOT remembered in the subroutine. You cannot use the PEN(n) function to read this event, but the other PEN(n) functions can be read for the event.

### Example:

```
10 ON PEN GOSUB 3000
20 PEN ON
...
2990 END
3000 REM LIGHT PEN ROUTINE
...
3990 RETURN
```

## ON PLAY Statement

**Format:** ON PLAY (n) GOSUB line

**Purpose:** Permits continuous background music during program execution.

**Remarks:** n is a numeric expression between 1 and 32.

line is the first line of the trapping routine. A line number of 0 disables the trap.

**Note** See the PLAY statement for further details.

The ON PLAY statement causes a branch to a specific subroutine when the number of notes in the Background Music buffer is less than n (specifically, when the number of notes in the buffer goes from n to n-1 stored notes.) Music is played in the background by a PLAY "MB ... statement. ON PLAY traps have no effect when music is running in the foreground (PLAY "MF ...).

A PLAY ON statement must be used to start ON PLAY trapping. PLAY OFF disables play event trapping, and the GOSUB is not executed or remembered. PLAY STOP suspends the event trapping. The GOSUB is not performed, but will be executed as soon as a PLAY ON statement is executed.

### Note

If the Background Music buffer contains less than n notes when the ON PLAY statement is first executed, the event trap will NOT be activated.

When an ON PLAY event trap occurs, an automatic PLAY STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine automatically performs a PLAY ON statement unless an explicit PLAY OFF was performed inside the subroutine.

Event trapping only happens when Vectra BASIC is running a program. When an error trap (resulting from an ON ERROR statement) occurs, all trapping is automatically disabled. This includes ERROR, COM(n), KEY(n), PEN(n), STRING(n) and TIMER statements.

You can use a RETURN line# statement in the trapping routine to return to a specific line number. However, you must exercise caution when using the RETURN statement in this manner. For example, any other GOSUBs, WHILEs, or FORs that were active when the trap occurred will remain active.

### Caution

n should be a number somewhat smaller than the Music Background buffer size. If it is only slightly less than the buffer size, event traps will occur frequently and will slow down the execution of your program.

### Example:

This example plays a music string in the background. When the number of notes in the Background Music buffer falls below 3, it branches to line 2000, where it adds the same string to the notes in the buffer.

```
100 LET LISTEN$ = "MB 1180 D2 P2 P8 L8 GGG L2 E-"
110 FATE$ = "P24 P8 L8 FFF L2 D"
120 PLAY LISTEN$ + FATE$
130 ON PLAY(3) GOSUB 2000
140 PLAY ON
...
1990 END
2000 PLAY LISTEN$ + FATE$
2010 RETURN
```

## ON STRIG Statement

**Format:** ON STRIG (n) GOSUB line

**Purpose:** Specifies the first line number of a subroutine to be performed when a joystick trigger is pressed.

**Remarks:** n is the number of the joystick trigger which is to be trapped. The values for n are:

- 0 A1 button
- 2 B1 button
- 4 A2 button
- 6 B2 button

line is the number of the first line of the subroutine. A line of zero disables the event trap.

An STRIG(n) ON starts ON STRIG event trapping. When joystick event trapping is active, Vectra BASIC checks between statements to see if the joystick trigger has been pressed. If it has been pressed, the GOSUB is performed.

If an STRIG(n) OFF has been executed, the GOSUB is not performed and is not remembered.

If an STRIG(n) OFF has been executed, the GOSUB is not performed immediately, but the event is remembered. The GOSUB will be performed as soon as an STRIG(n) ON statement is executed.

When an ON STRIG(n) event trap occurs, an automatic STRIG(n) STOP takes place, so recursive trapping cannot take place. The RETURN from the trapping subroutine automatically does an STRIG(n) ON statement unless an explicit STRIG(n) OFF was performed inside the subroutine.

## Note

An event which causes a trap is always "discarded". If you trap all four joystick buttons to the same subroutine line number, you cannot tell which joystick button caused the trap. If you need to know which button was pressed, you must use a different subroutine for each button.

Event trapping only occurs when Vectra BASIC is running a program, not in Direct Mode. Event trapping is disabled when an error trap occurs. This includes ERROR, COM(n), KEY(n), PEN(n), PLAY and TIMER statements.

You can use a RETURN line# statement in the trapping routine to return to a specific line number. However, you must exercise caution when using the RETURN statement in this manner. For example, any other GOSUBs, WHILEs, or FORs that were active when the trap occurred will remain active.

## Example:

This example traps both buttons on joystick A. Each of the buttons has its own subroutine.

```
10 ON STRIG(0) GOSUB 2000 : ON STRIG(4) GOSUB 2200
20 STRIG(0) ON: STRIG(4) ON
...
1990 END
2000 REM TRAP SUBROUTINE FOR BUTTON A-1
...
2190 RETURN
2200 REM TRAP SUBROUTINE FOR BUTTON A-2
...
2390 RETURN
```

## ON TIMER Statement

**Format:** ON TIMER(*n*) GOSUB *line*

**Purpose:** Causes an event trap every *n* seconds.

**Remarks:** *n* is a numeric expression that ranges between 1 and 86400 seconds (24 hours). Numbers outside this range produce an illegal function call error.

*line* is the beginning line number of the trap routine for TIMER.  
A line number of zero stops the timer trap.

A TIMER ON statement must be used to start an ON TIMER trap statement.

A TIMER OFF statement disables timer trapping. A TIMER STOP statement suspends TIMER event trapping. If a TIMER event occurs, the event is remembered, and the GOSUB is performed when a TIMER ON statement is executed.

### Note

See the TIMER statement for further details.

Event trapping only happens when Vectra BASIC is running a program. When an error trap (resulting from an ON ERROR statement) occurs, all trapping is automatically disabled. This includes all ERROR , COM(*n*) , and KEY(*n*) statements.

You can use a RETURN *line#* statement in the trapping routine to return to a specific line number. However, you must exercise caution when using the RETURN statement in this manner. For example, any other GOSUBs, WHILEs, or FORs that were active when the trap occurred will remain active.

### Example:

```
10 REM On each minute, display the time of
20 ON TIMER(60) GOSUB S000
30 TIMER ON
...
5000 REM Time message subroutine
5010 X = CSRLIN 'Save current row
5020 Y = POS(0) 'Save current column
5030 LOCATE 1,1 : PRINT TIME$;
5040 LOCATE X,Y 'Restore old row and column
5050 RETURN
```

## OPEN Statement

**Format 1:** OPEN (*dev*|*filename*) {FOR *mode*:} AS (*#*|*filename*|LEN+*rec*)

**Format 2:** OPEN *mode*2, (*#*|*filename*, (*dev*|*filename*) [*rec*])

**Purpose:** Grants access to a file or a character device for reading or writing.

**Remarks:** In Format 1, *mode* can be:

|        |                                                                                                                          |
|--------|--------------------------------------------------------------------------------------------------------------------------|
| INPUT  | for sequential input mode                                                                                                |
| OUTPUT | for sequential output mode                                                                                               |
| APPEND | for sequential output mode. Additionally, Vectra BASIC positions the file to the end of the data when you open the file. |

When you omit the *mode* parameter, the program assumes random access.

**Note** Even though *mode* is a string constant, you must not enclose the string in quotation marks.

In Format 2, *mode*2 can be:

|   |                                                                                        |
|---|----------------------------------------------------------------------------------------|
| I | for sequential input mode                                                              |
| O | for sequential output mode                                                             |
| A | for sequential append mode                                                             |
|   | Vectra BASIC positions the file pointer to the end of the file when you open the file. |
| R | for random input/output mode                                                           |

## Note

*mode*2 must be enclosed in quotes

Disc files allow all four modes. The APPEND or A mode can be used only with disc files, not with character devices.

*filename* is a string expression that names the file. Pathnames are permissible. It may include a file type (.xxx) and a drive specifier if the file is not on the current disc. When *filename* is a literal, you must enclose the string in quotation marks ("").

*filenum* is an integer expression that gives that file's identifying number. Its value may range from 1 to the maximum number of files allowed. The normal maximum setting is 5, but you may change this value with the /F: switch on the Vectra BASIC command line.

## Note

Other versions of BASIC allow different numbers of open files. If you are going to run your program under a different version of BASIC, you should check this parameter.

Once you assign a number to the file, Vectra BASIC associates this number to that file for as long as the file remains open. You use *filenum* when using other I/O statements with the file or device.

*rec* is an integer expression that sets the record length. You can define *rec* for random-access files. The default is 128 bytes. The value you use for *rec* must not exceed the value you set on the Vectra BASIC command line for the /S: switch when you initialized Vectra BASIC.

## Note

You may also set the maximum record length by using the /S option when initializing Vectra BASIC with the MS-DOS GW BASIC command. However, you cannot use this option with sequential files.

*dev* is a Vectra BASIC character device, one of the following:

KYBD:       the keyboard.  
SCRN:       the screen.  
LPT*n*:       printers. *n* can range from 1 to 3.  
COM*n*:       Communications port. *n* can be 1 or 2.

A special form of the OPEN statement exists for use with communications ports. See the OPEN "COM" Statement.

Character devices, such as printers, are opened and used in the same way as disc files. However, characters are not buffered by Vectra BASIC as they are for disc files. The record length is set to one.

### Note

You must ensure that your printer is properly configured before undertaking any printer operation. Failing to do so may result in an *Unprintable error in line #* message.

The two formats for the OPEN statement are interchangeable. Your program must execute an OPEN statement before you can use any of the following commands:

```
PRINT #, PRINT # USING, WRITE #  
INPUT #, INPUT#, LINE INPUT #  
FIELD #, GET # and PUT #
```

You must open a disc file or device before you can perform any read or write operation on it.

The OPEN statement allocates an I/O buffer to the file and determines the buffer's mode of access.

If *filename* does not exist in the specified directory on the disc, and the file is opened for append, output, or random access, the file is created and then opened. If *filename* does not exist and the file is opened for input, a **File not found** error results.

A random file can be opened with two different file numbers at the same time, and a sequential file can be opened for INPUT with two different file numbers at the same time.

You cannot open a sequential file for OUTPUT or APPEND with two different file numbers at the same time. In addition, if two files reside on different directories (on the same drive), but have the same name, you cannot have them both open at once. This is true even if you specify complete and obviously different paths; only the actual filename portion is considered. For example:

```
100 OPEN "A:\ACCOUNTS\MAY" FOR OUTPUT AS #1  
110 OPEN "A:\LEDGER\MAY" FOR OUTPUT AS #2
```

will produce the error message **File already open in 110**. You may have two files with the same name on different drives open at the same time. The following lines are legal:

```
100 OPEN "A:\ACCOUNTS\MAY" FOR OUTPUT AS #1  
110 OPEN "B:\LEDGER\MAY" FOR OUTPUT AS #2
```

**Examples:**

The first example writes a message to LPT1:

```
10 OPEN "0", #1, "LPT1:"
20 PRINT #1, "HELLO"
30 CLOSE #1
```

The next example opens the file MAIL.DAT so data is added to the end of the file:

```
10 OPEN "MAIL.DAT" FOR APPEND AS #1
```

**Note**

If you plan to compile your program, see the BASIC compiler manual for differences in the interpretive and compiled versions of this statement.

**OPEN "COM Statement**

**Format:**

```
OPEN "COM": {speed} [, {parity} [, {data} [, {stop} [, {RS} [, {PE}
[, {CS(n)}] [, {DS(n)}] [, {BIN} [, {ASC} [, {LF}]]]
[FOR mode] AS #] filenum [LEN=number]
```

**Purpose:**

Opens a communications file by allocating a buffer for I/O, similar to the way that an OPEN statement allocates a buffer for disc I/O.

**Remarks:**

*n* is the communication port number. Permissible values for *n* are 1 or 2.

*speed* is the transmit/receive baud rate in bits per second (bps). *speed* may equal 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800 or 9600. The default setting is 300 bps.

*parity* is a one-character abbreviation that specifies the parity method: S (space); O (odd); E (even); M (mark); or N (none). The default setting is E (even).

*data* has an integer value of 5, 6, 7 or 8. It gives the number of transmit/receive data bits. The default setting is 7.

*stop* sets the number of stop bits. Stop can be 1, 1.5 or 2. The default setting for 75 and 110 bps is a 2-bit stop. For all other speeds, the default setting is a 1-bit stop.

*mode* can be one of the following

- OUTPUT Specifies sequential output mode.
- INPUT Specifies sequential input mode.

If *mode* is omitted, the file is opened for random input/output. (You do not explicitly specify random mode.)

*filenum* is an integer that gives the file's identifying number. You use *filenum* with I/O statements.

*number* is the maximum number of bytes that can be read from the communication buffer when using **GET** or **PUT**. The default setting is 128 bytes.

The following options are available:

- LF** Sends a line feed character after each carriage return character.
- RS** suppresses **RTS** (Request to Send).
- CS(n)** control **CTS** (Clear to Send).
- DS(n)** controls **DSR** (Data Set Ready).
- CD(n)** controls **CD** (Carrier Detect).
- PE** included for compatibility only.
- BIN** opens the device in binary mode. This is the default mode.
- ASC** opens the device in ASCII mode.

The **RS**, **CS**, **DS**, and **CD** options control the status of the device control lines.

By default, **RS** is turned on when you issue an **OPEN "COM** statement. Including the **RS** option suppresses this action.

The **CS**, **DS** and **CD** options check the status of the device control lines. They can each accept an argument *n*, which specifies the number of milliseconds to wait before issuing a **Device Timeout** error. *n* can range from 0 to 65535. If *n* is zero, or if *n* is omitted, the line status of that option is not checked. The default values are: **CS**1000, **DS**1000 and **CD**0, waiting one second for Clear to Send and Data Set Ready. (If **RS** was specified, **RS**0 is the default.)

The **PE** option is included for compatibility with other versions of BASIC. In Vectra BASIC, **PE** has no effect. Vectra BASIC always checks parity. Other versions of BASIC check parity only when the **PE** option is included in the **OPEN "COM** statement.

In the **BIN** mode, tabs are not expanded to spaces, a carriage return is not forced at the end-of-line, and Control-Z is not treated as end-of-file. When the channel is closed, Control-Z will not be sent over the RS-232 line. The **BIN** option supercedes the **LF** option.

In the **ASC** mode, tabs are expanded, carriage returns are forced at the end-of-line, Control-Z is treated as end-of-file, and XON/XOFF protocol (if supported) is enabled. When the channel is closed, Control-Z will be sent over the RS-232 line.

When you are using communication files, you should set the **LF** parameter to permit printing to a serial line printer. When you specify **LF**, Vectra BASIC appends a line feed character after each carriage return character.

When you specify 8 data bits, you must set parity to none (that is, **N**).

Since the communication port is opened as a file, most disc I/O statements are valid for **COM** files. For example, **COM** sequential input statements are:

```
INPUT# filename
LINE INPUT# filename
INPUT#
```

**COM** sequential output statements are:

```
PRINT# filename
PRINT# filename USING
```

You must give an **OPEN "COM** statement before you can use a device for RS-232 communications.

Any syntax errors in the **OPEN "COM** statement cause a **Bad file name** error. However, Vectra BASIC gives no indication which parameter caused the error.

You must list the *speed*, *parity*, *data*, and *stop* parameters in the order they are shown in the format diagram. You may list the remaining options in any order, but they must follow these first four parameters.

### Examples:

The first statement uses all the default settings. That is, the speed is 300 bps and there is even parity, seven data bits, and one stop bit.

```
10 OPEN "COM1:" AS #1
```

The next statement sets the speed to 1200 bps and parity to odd.

```
10 OPEN "COM1:1200,0" AS #1
```

The following program implements a very dumb terminal that uses XON/XOFF protocol.

```
10 REM Dumb terminal for modem using XON/XOFF
20 DEFINT A-Z
30 CLOSE
40 FALSE = 0 : TRUE = NOT FALSE
50 XOFF$ = CHR$(19) : XON$ = CHR$(17)
60 ECHO = FALSE
70 OPEN "COM1:300,N,8,1" AS #1
80 PAUSE = FALSE
90 PRINT "In: super dumb 'T to Terminate ";
95 PRINT "'E to toggle Echo"
100 B$ = INKEY$ : Did user type anything?
110 IF B$ = "" THEN 150
120 IF B$ = CHR$(20) THEN GOTO 280
130 IF B$ = CHR$(5) THEN GOSUB 300 : GOTO 150
140 PRINT #1, B$ : IF ECHO THEN PRINT B$
150 IF EOF(1) THEN 100 ' Any data in modem?
160 REM If modem is filling up, next statement
161 REM stops transmission
170 IF LOC(1) > 45 THEN PAUSE = TRUE : PRINT #1, XOFF$
180 AS = INPUT$(LOC(1), #1) ' Get data from modem
190 LFP = 0
195 REM Turn line feeds into blanks
200 LFP = INSTR(LFP + 1, AS, CHR$(10))
210 IF LFP > 0 THEN MID$(AS, LFP, 1) = " " : GOTO 200
220 PRINT AS :
230 IF LOC(1) > 0 THEN 170 ' More data in modem?
240 REM Next statement starts receiving again
250 IF PAUSE THEN PAUSE = FALSE : PRINT #1, XON$ :
260 GOTO 100
270 REM Terminate program
280 CLOSE
290 END
300 REM Toggle Echo
310 ECHO = NOT ECHO
320 RETURN
```

## OPTION BASE Statement

### Format:

```
OPTION BASE #
```

### Purpose:

Sets the minimum value for array subscripts.

### Remarks:

# may be either 1 or 0.

Vectra BASIC normally numbers arrays from a base of zero. When you want an array index to begin at 1, you must use the OPTION BASE statement.

If you decide to use the OPTION BASE statement, you must include it within your program before you define or use any arrays.

Chained programs may have an OPTION BASE statement if no arrays are passed between them or the specified base is identical in the chained programs. The chained program will inherit the OPTION BASE value of the chaining program.

### Example:

This example sets up a string array with ten elements (1..10) and a numeric array with 20 elements (1..20):

```
10 OPTION BASE 1
20 DIM LNAME$, ID(20)
```

## OUT Statement

**Format:** OUT *i,j*

**Purpose:** Sends a byte to the specified output port.

**Remarks:** *i* is an integer expression that ranges between 0 and 65535. It is a microprocessor port number.

### Note

The output port is a microprocessor port. It does not refer to your computer's datacomin (or peripheral) ports.

Port locations are hardware-dependent, and may not be the same on other computers.

*j* is an integer expression that ranges between 0 to 255. It is the byte of data that you want to send. For example, a zero sets all eight bits to zeroes while 255 sets all eight bits to ones.

OUT is the complementary command to the INP function.

### Example:

This example uses OUT to change colors on the screen. In medium resolution, it changes the background color; in high resolution it changes the foreground colors. See the COLOR statement for a list of colors. (Use of this statement can produce non-standard combinations of brightness in medium resolution.)

```
100 OUT (&H3D9),9
```

## PAINT Statement

**Format:** PAINT (*x,y*) ( *i,fill* ) (*boundary*) ( *background* )

**Purpose:** Fills an area on the screen with the selected color or pattern.

**Remarks:** *x,y* are the coordinates where painting begins. In medium resolution, *x* can range from 0 to 319, and in high resolution, *x* can range from 0 to 639. In both graphics modes, *y* can range from 0 to 199. PAINT is only valid in a graphics mode.

If *x,y* is inside a graphics figure, the figure will be filled. If *x,y* is outside a graphics figure, the screen background will be painted with the selected color or pattern.

*fill* is the fill color or pattern. When *fill* is a numeric expression, it fills with a solid color; when *fill* is a string, the area is filled with a "tiling" pattern.

*boundary* is the color of the edges of the figure to be filled.

In medium-resolution graphics, *boundary* and *fill* can be numeric expressions returning a value from 0 to 3. The value selects a color from the palette chosen by a COLOR statement. When you omit either parameter, Vectra BASIC uses the default color 3 from the palette selected by a COLOR statement.

In high resolution graphics, *fill* and *boundary* can be 0 or 1. 0 selects the background color (always black) and 1 selects the foreground color, as selected by a COLOR statement.

**Note**

*boundary* is the color you specified in a LINE, CIRCLE or DRAW command. If you specify the wrong boundary color, PAINT will blot out the figure and continue to fill until it finds a boundary of the specified color, or until it fills the entire screen.

*background* is a string expression that returns a single character. It is used in tiling to overwrite an existing pattern or a solid color.

When you omit this parameter, the default is CHR\$(0)

**Note**

You may use the PAINT statement to fill any graphics figure, but painting jagged edges or very complex figures may result in an out of memory error. To prevent this from happening, you should use the CLEAR statement to increase the amount of available stack space.

**Tiling**

When fill is a string formula, Vectra BASIC uses this string as a tiling mask to set pixels on the screen.

The pattern for tiling is set by a string of characters, in the form PAINT (x,y),CHR\$(n)..., where n ranges from 0 to 255. It is often easiest to construct the patterns using hex values. Using hex values, the tiling pattern would be represented as CHR\$(Hmn), where mn is a hex value between 00 and FF. There may be up to 64 characters in a tiling pattern.

The tiling process uses the arrangement of bits (ones and zeroes) in a character to form the pattern. For example, #H55 is represented in binary as 01010101. When CHR\$(#H55) is used as a tiling pattern on the high resolution screen, every other pixel in the filled area is set, and every other one is left off, filling the area with a pattern of very fine lines.

When the tiling mask is more than one character long, each byte is used in succession. The bytes of the tile string are always aligned horizontally and vertically so that the tile pattern is replicated uniformly over the entire screen (as if you specified PAINT(0,0)...).

| x increases -->  |   |   |   |   |   |   |   |   |                  |
|------------------|---|---|---|---|---|---|---|---|------------------|
| bit of tile byte |   |   |   |   |   |   |   |   |                  |
| x,y              | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | Tile byte number |
| 0.0              | x | x | x | x | x | x | x | x | 1                |
| 0.1              | x | x | x | x | x | x | x | x | 2                |
| 0.2              | x | x | x | x | x | x | x | x | 3                |
| ...              |   |   |   |   |   |   |   |   |                  |
| 0.63             | x | x | x | x | x | x | x | x | 64 (maximum)     |

You can paint the high resolution screen with X's by using the following statement:

```
PAINT(320,100),CHR$(#H81)*CHR$(#H42)*CHR$(#H24)
*CHR$(#H18)*CHR$(#H18)*CHR$(#H24)*CHR$(#H42)
*CHR$(#H81)
```

Executing this statement has the following effect:

| x increases --> |   |   |   |   |   |   |   |   |                         |
|-----------------|---|---|---|---|---|---|---|---|-------------------------|
| 0.0             | x |   |   |   |   |   |   |   | CHR\$(#H81) Tile byte 1 |
| 0.1             |   | x |   |   |   |   |   |   | CHR\$(#H42) Tile byte 2 |
| 0.2             |   |   | x |   |   |   |   |   | CHR\$(#H24) Tile byte 3 |
| 0.3             |   |   |   | x |   |   |   |   | CHR\$(#H18) Tile byte 4 |
| 0.4             |   |   |   |   | x |   |   |   | CHR\$(#H18) Tile byte 5 |
| 0.5             |   |   |   |   |   | x |   |   | CHR\$(#H24) Tile byte 6 |
| 0.6             |   |   |   |   |   |   | x |   | CHR\$(#H42) Tile byte 7 |
| 0.7             |   |   |   |   |   |   |   | x | CHR\$(#H81) Tile byte 8 |

The following program demonstrates the patterns created by tiling patterns CHR\$(1) through CHR\$(16).

```

10 SCREEN 1
20 CLS
30 REM Make some boxes
40 FOR X=0 TO 319 STEP 40
50 LINE (X,0)-(X,199)
60 NEXT X
70 LINE (319,0)-(319,199)
80 LINE (0,75)-(319,75)
90 LINE (0,105)-(319,105)
100 T=1:REM T is tile pattern counter
110 REM Fill first row of boxes
120 FOR J=10 TO 319 STEP 40
130 PAINT (J,10),CHR$(T)
140 T=T+1
150 NEXT J
160 REM Fill second row of boxes
170 FOR J=10 TO 319 STEP 40
180 PAINT (J,120),CHR$(T)
190 T=T+1:Increment counter
200 NEXT J
210 REM Blank out the center space
220 LINE (0,76)-(319,104),0,BF
230 REM Print Binary values for Tiles
240 LOCATE 11,1
250 PRINT "0001 0010 0011 0100 0101 0110 0111 1000"
260 PRINT
270 PRINT "1001 1010 1011 1100 1101 1110 1111 100000"

```

On the medium resolution screen, creating predictable tiling patterns is more complex. Each pixel on the medium resolution screen is represented by two bits of information. A tiling character in medium resolution sets 4 pixels, with a pair of bits in the tiling pattern determining the pixel's color.

The following diagram depicts the relationship between decimal values and pixel color.

| Pixel       | 1   | 2  | 3  | 4  |
|-------------|-----|----|----|----|
| Bit value   | 128 | 64 | 32 | 16 |
| Color value | 2   | 1  | 2  | 1  |
|             |     |    | 8  | 4  |
|             |     |    | 2  | 1  |
|             |     |    | 1  | 2  |
|             |     |    | 1  | 1  |

The tiling character CHR\$(128) produces color 2, CHR\$(64) produces color 1, and CHR\$(192) produces color 3 in the first pixel set by each tiling character.

Here are examples of some other values using colors from palette 1:

| Pixel                   | 1     | 2  | 3  | 4  |
|-------------------------|-------|----|----|----|
| Bit value               | 128   | 54 | 32 | 16 |
| Color value             | 2     | 1  | 2  | 1  |
| Result:                 | CHR\$ |    |    |    |
| solid magenta           | 170   | 1  | 0  | 1  |
| solid cyan              | 85    | 0  | 1  | 0  |
| cyan/magenta pinstripe  | 102   | 0  | 1  | 0  |
| black and white stripes | 15    | 0  | 0  | 0  |
|                         |       | 1  | 1  | 1  |
|                         |       | 1  | 1  | 1  |

On the medium resolution screen, the statement above which produces uniform X's in high resolution produces X's composed of all three colors on the medium resolution screen.

Normally, Vectra BASIC stops tiling when it encounters two consecutive lines that match the tiling pattern. The *background* parameter is used in those rare cases when you need to tile an area that is already filled with a solid color that is used in your tiling pattern, or an area that is tiled with two consecutive lines that match a lines in your tiling pattern.

This example demonstrates the use of the *background* parameter.

```

10 T$=CHR$(102)*CHR$(102)*CHR$(170)
20 CLS:SCREEN 1:COLOR 0,1:KEY OFF
30 LOCATE 2,1:PRINT "TITLE OVER SOLID:"
40 LOCATE 4,1:PRINT"NO BACKGROUND:"
50 LINE (20,40)-(80,90),2,BF
60 PAINT (30,50),CHR$(40)*CHR$(170),0
70 LOCATE 14,1:PRINT"WITH BACKGROUND:"
80 LINE (20,130)-(80,180),2,BF
90 PAINT (30,140),CHR$(40)*CHR$(170),0,CHR$(170)
100 LOCATE 2,21:PRINT "TILE OVER TILE:"
110 LOCATE 4,21:PRINT"NO BACKGROUND:"
120 LINE (180,40)-(240,90),1,B
130 PAINT (190,50),CHR$(102)*CHR$(102)
140 PAINT (190,50),T$
150 LOCATE 14,21:PRINT"WITH BACKGROUND:"
160 LINE (180,130)-(240,180),1,B
170 PAINT (190,140),CHR$(102)*CHR$(102)
180 PAINT (190,140),T$,CHR$(102)
190 END

```

### Example:

```

10 REM TaxMan demonstration showing tiling with
    PAINT
20 ' and animation through PUT and GET
30 DIM PM0$(250), PM1$(250), PM2$(250), PM3$(250)
40 PI = 3.1415926#
50 DEF FNR(D) = D*PI / 180!
60 SCREEN 1
70 CLS
80 GET (279,30)-(319,70), PM0$
90 A = 45 : B = 315 : GOSUB 300
100 GET (279,30)-(319,70), PM1$
110 A = 22 : B = 338 : GOSUB 300
120 GET (279,30)-(319,70), PM2$
130 A = 1 : B = 359 : GOSUB 300
140 GET (279,30)-(319,70), PM3$
150 CLS
160 FOR X = 40 TO 320 STEP 10
170   CIRCLE (X,70), RND*10
180   PAINT (X,70), CHR$(AHAA)
190 NEXT X
200 FOR X = 0 TO 279 STEP 4
210   PUT (X,50), PM1$, PSET
220   FOR J = 1 TO 50 : NEXT J
230   PUT (X,50), PM2$, PSET
240   FOR J = 1 TO 65 : NEXT J
250   PUT (X,50), PM3$, PSET
260   FOR J = 1 TO 50 : NEXT J
270   PUT (X,50), PM0$, PSET
280 NEXT X
290 END
300 REM ***** Subroutine Section *****
310 CLS
320 CIRCLE (300,50),20,..FNR(-A),FNR(-B)
330 CIRCLE (303,37),3
340 PAINT (300,47), CHR$(AHAA) + CHR$(AH55) +
    CHR$(AHCC) + CHR$(AH33)
350 RETURN

```

## PEEK Function

**Format:**            PEEK(i)

**Action:**            Returns the byte read from memory location *i*.

The result is a decimal integer that ranges between 0 (eight zeros) to 255 (eight ones).

*i* must be within the range of -32768 to 65535. (It is an offset from the current segment, which you set with the DEF SEG statement.) When the function returns a negative value, you should add 65536 to that value to obtain the actual address.

PEEK is the complementary function to the POKE statement.

**Example:**            A = PEEK(456789)

## PEN Statement

**Format:**  
PEN ON  
PEN OFF  
PEN STOP

**Purpose:**            Enables, disables or suspends ON PEN event trapping and PEN lightpen reading.

**Remarks:**  
PEN ON            Enables lightpen reading and event trapping  
PEN OFF           Disables lightpen reading and event trapping  
PEN STOP          Suspends lightpen reading and event trapping.

You must use a PEN ON statement before you can read the light pen with the PEN function or trap the use of the light pen with an ON PEN statement.

PEN OFF disables the lightpen. Programs execute faster when the pen is off, so turn off the pen for portions of a program when it is not needed. It also disables pen trapping.

PEN STOP suspends pen reading and trapping, but remembers a PEN event, and the GOSUB is performed as soon as event trapping and reading is enabled with a PEN ON statement

### Note

See the PEN function for a full explanation of pen reading. See ON PEN for an explanation of pen trapping.

### Examples:

```
10 PEN ON : REM ENABLE THE LIGHT PEN
20 ON PEN GOSUB 2000
...
1000 REM DISABLE PEN FOR THIS PART
1010 PEN OFF
...
2000 REM PEN TRAP ROUTINE
...
3000 RETURN
```

## PEN(n) Function

**Format:** PEN(*n*)

**Action:** Returns information about the position of the lightpen.

The pen must first be enabled by a PEN ON statement. The instruction *v*=PEN(*n*) reads different information about the use of the lightpen, depending on the value of *n*. This chart lists the possible values for *n*, and the information that is stored in *v*.

| <i>n</i> | Information returned in <i>v</i>                                                                                           |
|----------|----------------------------------------------------------------------------------------------------------------------------|
| 0        | <i>v</i> = -1 if pen has been touched to the screen since the last PEN(0) poll. <i>v</i> = 0 if the pen has not been used. |
| 1        | <i>v</i> = the <i>x</i> pixel coordinate where the pen was last pressed.                                                   |
| 2        | <i>v</i> = the <i>y</i> pixel coordinate where the pen was last pressed.                                                   |
| 3        | <i>v</i> = -1 if pen is currently down; <i>v</i> = 0 if the pen is up.                                                     |
| 4        | <i>v</i> = the last known valid <i>x</i> pixel coordinate.                                                                 |
| 5        | <i>v</i> = the last known valid <i>y</i> pixel coordinate.                                                                 |
| 6        | <i>v</i> = the character row position where the pen was last pressed.                                                      |
| 7        | <i>v</i> = the character column position where the pen was last pressed.                                                   |
| 8        | <i>v</i> = the last known character row where the pen was last pressed.                                                    |
| 9        | <i>v</i> = the last known character column where the pen was last pressed.                                                 |

*n* values of 0 and 3 test for the use of the pen, and are used both in graphics and text modes. PEN(0) tests to see if the pen was used since the last poll, and PEN(3) tests to see if the pen is currently depressed.

*n* values of 1, 2, 4 and 5 are used in graphics modes. PEN(1) and PEN(2) return the most recent coordinates; PEN(4) and PEN(5) return the coordinates from the previous pen press. In medium resolution, the *x* coordinate that is returned by a PEN(1) or PEN(4) function can range from 0 to 319; in high resolution, the *x* coordinate can range from 0 to 639. In both graphics modes, the *y* coordinate reported by a PEN(2) or PEN(5) function can range from 0 to 199.

*n* values of 6, 7, 8 and 9 are used in text mode. PEN(6) and PEN(7) return the most recent coordinates; PEN(8) and PEN(9) return the coordinates from the previous pen press. The row value reported by PEN(6) or PEN(8) can range from 1 to 25. The column value reported by PEN(7) or PEN(9) can range from 1 to 40 at WIDTH 40, or 1 to 80 at WIDTH 80.

### Note

Light pen positions near the edges of the screen may report inaccurate values on some monitors.

### Example:

This example produces an endless loop which prints the pen position:

```
10 CLS
20 PEN ON
30 P=PEN(3)
40 LOCATE 1,1: PRINT "PEN IS ";
50 IF P THEN PRINT "DOWN" ELSE PRINT "UP"
60 GOTO 30
```

### Note

See the PEN statement for more details.

## PLAY Statement

### Format 1:

PLAY ON  
PLAY OFF  
PLAY STOP

### Purpose:

Enables, disables, or suspends PLAY event trapping.

### Remarks:

PLAY ON enables event trapping by the ON PLAY statement.

When a PLAY OFF statement has disabled even trapping, the GOSUB in the ON PLAY statement is not performed, and it is not remembered.

If a PLAY STOP statement is executed, the GOSUB is not performed immediately, but it is remembered, and will be performed when a PLAY ON statement is executed.

### Note

See the ON PLAY statement for further details.

### Format 2:

PLAY *string*

### Purpose:

Plays the specified notes.

### Remarks:

*string* is made up of the following commands. The commands affect the tone of the notes, their duration, and the tempo of the music. Music can be played in the background while other Vectra BASIC instructions are executed, or can play in the foreground.

The letter *n* in the commands can be an integer, or it can be a variable name. Variables must be typed `%variable`. Spaces in *string* are ignored.

### Tone

ON

Selects the octave for the subsequent notes. There are seven octaves; *n* may range from 0 to 6. The default octave is 4, middle C is the first note of octave 3. The notes in an octave are in the order CDEFGAB.

A-G

Plays the specified note. To play sharps and flats, append one of the following:

\* or # Sharp  
- Flat

Notes followed by sharps and flats which do not have corresponding black keys on a piano will result in an illegal function call error message.

NN

Plays note *n*. *n* may range from 0 to 64. *n* = 0 means a rest. There are 7 octaves of 12 notes each.

> *n*

Go up one octave, and play note *n*. If the octave is 6, this command has no effect, since the octave cannot be greater than 6.

< *n*

Go down one octave, and play note *n*. Octave cannot be less than 0. If the octave is already 0, this command has no effect.

### Duration

L *n*

Sets the length of the following notes. *n* can range from 1 to 64. L 1 is a whole note; L 4 is a quarter note.

To change the length for only one note, follow the note with the length. L 64 A and A 64 both play sixty-fourth notes. The first command affects all subsequent notes, up to another L command; the second form affects only the A, and leaves the previous L command in effect.

P *n*

Sets the length of a pause. *n* can range from 1 to 64. (period) A period after a note causes it to play one-half again as long as normal. A quarter-note followed by a period (B .) plays for the duration of a quarter-note plus and eighth-note. An additional period (B . .) prolongs the note the length of a one-sixteenth note.

|                  |                                                                                                                                                                                                                                                                               |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>MM</b>        | Plays notes at "music normal". Each note plays for 7/8 of the time determined by the length L.                                                                                                                                                                                |
| <b>ML</b>        | Plays notes "legato". Each note plays for the full time set by L.                                                                                                                                                                                                             |
| <b>MS</b>        | Plays "staccato" notes. Each note plays for 3/4 of the time set by L.                                                                                                                                                                                                         |
| <b>Tempo</b>     |                                                                                                                                                                                                                                                                               |
| <b>T n</b>       | Sets the tempo. <i>n</i> is set to the number of quarter notes in a minute. <i>n</i> can range from 64 to 255. The default tempo is 120.                                                                                                                                      |
| <b>Operation</b> |                                                                                                                                                                                                                                                                               |
| <b>MF</b>        | Plays music (and notes from the SOUND statement) in the foreground. Each note or sound will not play until the last sound is finished. Execution of Vectra BASIC statements pauses at PLAY or SOUND statements until all the notes have started. This is the default setting. |
| <b>MB</b>        | Plays music (and notes from the SOUND statement) in the background. Notes and sounds are placed in a buffer, and the music plays while subsequent GW BASIC statements are executed.                                                                                           |
| <b>Substring</b> |                                                                                                                                                                                                                                                                               |
| <b>x</b>         | Executes a substring. The name of the substring is appended to x.                                                                                                                                                                                                             |
| <b>Note</b>      | The format is PLAY "XSTRING\$; ...". The semicolon is required.                                                                                                                                                                                                               |

**Examples:**

This example plays 3 notes, then changes the octave and plays the same series of notes.

```
PLAY "G F A > GFA <<GFA"
```

This program plays the beginning of the first movement of Beethoven's Fifth Symphony

```
100 LET LISTEN$ = "T180 D2 P2 P8 L8 GGG L2 E-"
110 FATE$ = "P24 P8 L8 FFF L2 D"
120 PLAY LISTEN$ + FATE$
```

Line 120 could have used the x substring function. This is the syntax:

```
120 PLAY "XLISTEN$; XFATE$;"
```

## PLAY(n) Function

**Format:**      `PLAY (n)`

**Action:**      Returns the number of notes in the Background Music queue.  
*n* is a dummy argument and may be any value. `PLAY(n)` returns 0 when music is playing in Music Foreground mode.

**Example:**

```
100 PLAY "MB Q2 L4 GGA"  
110 J = PLAY (N)  
120 IF J < 2 PRINT "THE END"
```

## PMAP Function

**Format:**      `P = PMAP (x, n)`

**Action:**      Maps Physical Coordinates (PC) to World Coordinates (WC) and vice versa.

*x* is the coordinate of the mapped point.

*n* takes on a value that ranges between 0 and 3:

- 0      Maps the World Coordinate *x* to the Physical Coordinate *x*
- 1      Maps the World Coordinate *y* to the Physical Coordinate *y*
- 2      Maps the Physical Coordinate *x* to the World Coordinate *x*
- 3      Maps the Physical Coordinate *y* to the World Coordinate *y*

`PMAP` translates coordinates between the world-coordinate system defined in the `WINDOW` statement to the physical-coordinate system defined by the `VIEW` statement.

`PMAP(x, 0)` and `PMAP(x, 1)` map values from the world coordinate system to the physical coordinate system.  
`PMAP(x, 2)` and `PMAP(x, 3)` map values from the physical coordinate system to the world coordinate system.

**Examples:**      After the statements `SCREEN 1 : WINDOW (-1, -1) - (1, 1)` execute, these translations can be made:

`PMAP(-1, 0)` returns the PC *x* value of 0.

`PMAP(-1, 1)` returns the PC *y* value of 199.

`PMAP(1, 0)` returns the PC *x* value of 319.

`PMAP(1, 1)` returns the PC *y* value of 0.

## POINT Function

### Format 1:

`u = POINT(x, y)`

### Action:

Reads the color value of a specific pixel on a graphics screen. This command is valid only in a graphics mode.

(x, y) are the absolute coordinates of a screen pixel. Relative coordinates are illegal.

In medium resolution mode, x can equal 0 to 319; in high resolution, x can equal 0 to 639. y can range from 0 to 199 in both graphics modes.

In medium resolution mode, POINT returns 0 - 3. 0 is the background color; 1-3 match the colors from the palette selected with a COLOR statement. In high resolution mode, POINT returns 0 for the background color and 1 for the foreground color.

The POINT function may also take the following form:

### Format 2:

`u = POINT (n)`

### Action:

Returns either the actual physical coordinates of the "last referenced" graphics point, or the World Coordinates for that point.

n Value returned

0 The current physical x coordinate

1 The current physical y coordinate

2 The current x World Coordinate, if a WINDOW statement is active.

3 The current y World Coordinate, if a WINDOW statement is active.

If no WINDOW is in effect, values of 2 and 3 have the same effect as values of 0 and 1.

Please refer to the description of the WINDOW statement for further details.

### Example:

```
10 REM Invert the current state of a point
20 IF POINT(ROW, COL) <> 0 THEN PRESET(ROW, COL)
   ELSE PSET(ROW, COL)
```

## POKE Statement

**Format:** POKE *address*, *data*

**Purpose:** Writes a byte of information into a memory location.

**Remarks:** *address* is an integer expression for the address of the memory location to be poked. (It is an offset from the current segment, which you set with the DEF SEG statement.) The value must be within the range of 0 to 65535.

*data* is an integer expression for the data to be poked. It must be within the range of 0 (which would set all eight bits to zeroes) to 255 (which would set all eight bits to ones).

PEEK is the complementary function to POKE. PEEK's argument is an address from which a byte of information is read.

You can use PEEK and POKE for efficiently storing data, loading assembly-language subroutines, and passing arguments and results to and from assembly-language subroutines.

### Caution

Vectra BASIC does not check the address. Therefore, use this statement with extreme care so you do not inadvertently overwrite meaningful data, MS-DOS, or the GW Basic interpreter.

### Example:

This example places hex value FF (decimal 255, or a byte with 1's in all eight positions) into the Data Segment relative memory location at hex 5A00:

```
10 POKE 4H5A00, 4HFF
```

## POS Function

**Format:** POS(*i*)

### Action:

Returns the cursor's current column position. The leftmost column is position number 1. The rightmost column is position number 40 or 80, depending on the SCREEN mode or WIDTH setting.

You may use the C\$RLIN function to return the cursor's current line position.

0 is a dummy argument.

See also the LPOS function and the WIDTH and SCREEN statements.

### Example:

```
IF POS(0) > 60 THEN PRINT CHR$(7)
```

**PRESET Statement**

**Format:**     PRESET( STEP )( *x*, *y* ) ( *color* )

**Purpose:**     Changes the color of a given pixel.

**Remarks:**   This command is valid only in a graphics mode.

*x* and *y* specify which pixel you want to set.

In medium resolution mode, *x* can equal 0 to 319; in high resolution, *x* can equal 0 to 639. *y* can range from 0 to 199 in both graphics modes.

*color* specifies which color to use. In medium resolution, 0 selects the background color. 1-3 select colors from the palette chosen by the COLOR statement. In high resolution, 0 selects the background color and 1 selects the foreground color as set by a COLOR statement.

When you omit *color*, Vectra BASIC uses the background color. PRESET works exactly like PSET, except that PSET uses the default foreground color if *color* is not specified.

You may give the coordinates in absolute or relative form. Using relative form requires the STEP option:

STEP ( *xoff*/*set*, *yoff*/*set* )

If you give an out-of-range coordinate, Vectra BASIC ignores the command. However, no error message is provided.

**Example:**

This example draws a line from (0,0) to (100, 100) and then erases that line by overwriting it with the background color:

```
5 REM Draw a line from (0,0) to (100,100)
6 SCREEN 1
10 FOR COORD = 0 TO 100
20   PSET (COORD, COORD), 3
30 NEXT
35 REM Now erase that line
40 FOR COORD = 0 TO 100
50   PSET STEP (-1,-1)
60 NEXT
```

## PRINT Statement

**Format:**

PRINT (*list of expressions*)

**Purpose:**

Copies data to the computer screen.

**Remarks:**

*list of expressions* is a list of numeric and/or string expressions. You must separate multiple items with commas, blanks, or semicolons and enclose any string constants between quotation marks.

Including *list of expressions* prints the values of those expressions on the screen.

Omitting *list of expressions* prints a blank line.

**Print Positions:** The punctuation symbols that separate the listed items determine the position where Vectra BASIC prints each item.

Vectra BASIC divides the line into print zones of 14 spaces each. Within *list of expressions*, a comma prints the next value at the beginning of the next zone. A semicolon prints the next value immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

When a comma or semicolon ends the *list of expressions*, the next PRINT statement continues printing on the same line, spacing accordingly. If the list ends with no comma or semicolon, Vectra BASIC ends the line by printing a carriage return character. (That is, it advances the cursor to the next line.)

When the printed line exceeds the width of the screen, Vectra BASIC wraps the line to the next physical line and continues printing.

For numbers, Vectra BASIC reserves the first character position for a numeric sign. It precedes positive numbers with a space. It precedes negative numbers with a minus sign. Vectra BASIC always prints a space as a separator after any number.

You may enter a question mark (?) as an abbreviation for the word PRINT in a PRINT statement. When Vectra BASIC lists the program, it automatically replaces the question mark with the reserved word PRINT.

To send output to a line printer, use the LPRINT and LPRINT USING statements.

### Note

When single-precision numbers can be represented with 7 or fewer digits in unscaled format no less accurately than they can be represented in scaled format, Vectra BASIC prints the numbers using unscaled format (either integer or fixed point). For example, Vectra BASIC prints 1E-7 as .0000001 whereas it prints 1E-8 as 1E-08.

When double-precision numbers can be represented with 16 or fewer digits in unscaled format no less accurately than they can be represented in scaled format, Vectra BASIC prints the numbers using the unscaled format. For example, Vectra BASIC prints

1D-16 as .000000000000000001 whereas it prints 1D-17 as 1D-17.

### Examples:

The commas in the following PRINT statement prints each successive value at the next print zone:

```
10 X = 5
20 PRINT X*S, X-S, X*S, X/S
30 END
RUN
10      0      25      1
Ok
```

In the following program segment, the semicolon at the end of line 20 prints the information from lines 20 and 30 on the same line. Line 40 prints a blank line before the next prompt:

```
10 INPUT X
20 PRINT X "SQUARED IS " X^2 "AND ";
30 PRINT X "CUBED IS " X^3
40 PRINT
50 GOTO 10
RUN
9 SQUARED IS 81 AND 9 CUBED IS 729
? 21 [Enter]
21 SQUARED IS 441 AND 21 CUBED IS 9261
? [CTRL] [Break]
```

In the following example, the semicolons in the PRINT statement print each value immediately after the preceding value. Remember, positive numbers are preceded by a space, and all numbers are followed by a space. Line 40 uses the question mark as an abbreviation for PRINT:

```
10 FOR X = 1 TO 5
20 J = J + 5
30 K = K + 10
40 ?J;K;
50 NEXT X
RUN
5 10 10 20 15 30 20 40 25 50
Ok
```

## PRINT USING Statement

**Format:** PRINT USING *stringexp*; *list of expressions*

**Purpose:** Uses a specified format to print strings or numbers. You normally use this statement when writing reports where the appearance of the document is critical.

**Remarks and Examples:** *list of expressions* contains the string or numeric expressions that you want to print. You must separate the items in the list with commas or semicolons.

*stringexp* is either a string constant or a string variable that contains special formatting characters. These formatting characters (see below) determine the field and format of the printed strings or numbers.

When entering program lines, you may use a question mark (?) as an abbreviation for the reserved word PRINT. Vectra BASIC automatically replaces this symbol with PRINT when you list the program.

### String Fields:

When you use the PRINT USING statement to print strings, you may select one of three characters to format the string field:

! An exclamation point limits printing to the first character in the string.

\n spaces \

Two back slash characters separated by n spaces prints that number of characters (that is,  $n + 2$ ). For example, typing just the backslashes prints two characters; typing one space between the backslashes prints three characters; and so on. When the field is longer than the string, Vectra BASIC left-justifies the string within the field and pads the remainder of the field with spaces. Consider this example:

```
10 A$ = "LOOK" : B$ = "OUT"
20 PRINT USING "1"; A$:B$
30 PRINT USING "\ \ "; A$:B$
40 PRINT USING "\ \ "; A$:B$;"!!"
RUN
LOOKOUT
LOOK OUT !!
OK
```

! An ampersand specifies a variable length string field. Using this formatting character echoes the string exactly as you entered it.

```
10 A$ = "LOOK" : B$ = "OUT"
20 PRINT USING "1"; A$;
30 PRINT USING "8"; B$
RUN
LOOK
OUT
OK
```

### Numeric Fields:

When printing numbers with the PRINT USING statement, you may use the following special characters to format the numeric field.

• The number sign signifies a digit position. Vectra BASIC fills in all requested digit positions. When a number has fewer digits than the positions specified, Vectra BASIC right-justifies the number in the field (that is, leading unused positions are replaced with spaces).

You may insert a decimal point at any position within the field. When the format string specifies that a digit should appear before the decimal point, Vectra BASIC always prints a digit (0 if necessary). Vectra BASIC also rounds numbers as required to fit the format.

Consider these examples:

```
PRINT USING "###.##"; .78
0.78
```

```
PRINT USING "###.##"; 987.654
987.65
```

```
PRINT USING "###.## "; 10.2, 5.3, 66.789, .234
10.20 5.30 66.79 0.23
```

In the last example, the three spaces at the end of the format string provide spacing between the printed values.

• A plus sign at the beginning or end of the format string prints the sign of the number (plus or minus) before or after the number, depending upon the placement of the plus sign in the format string.

```
PRINT USING "+.##.## "; -68.95, 2.4, 55.6, -.9
-68.95 +2.40 +55.60 -0.90
```

- A minus sign at the end of the format field prints a trailing minus sign after negative numbers.

```
PRINT USING "###.##- "; -68.95, 22.449, -7.01
68.95- 22.45 7.01-
```

\*\*. A double asterisk at the beginning of the format string replaces leading spaces with asterisks. The double asterisk also reserves two more digit positions.

```
PRINT USING "**###.## "; 12.39, -0.9, 765.1
*12.4 *-0.9 765.1
```

\$\$ A double dollar sign prints a dollar sign to the immediate left of the formatted number. The double dollar symbol reserves two more digit positions, one of which is the dollar sign. You cannot use the exponential format in conjunction with \$\$.

Furthermore, you can print negative dollar amounts only if the minus sign trails to the right.

```
PRINT USING "$###.##-"; 456.78, -45.54
$456.78 $45.54-
```

\*\*\* Placing \*\*\* at the beginning of a format string combines the effects of the two previous symbols. Vectra BASIC replaces leading spaces with asterisks and prints a dollar sign before the number. Additionally, \*\*\* reserves three digit positions, one of which is used for the dollar sign.

```
PRINT USING "***###.##"; 2.34
***$2.34
```

. A comma that appears to the left of the decimal point in a formatting string prints a comma as a thousands separator. When the comma appears at the end of the formatting string, the comma is printed following the number. The comma represents another digit position. It has no effect when used with the exponential format (^^^).

```
PRINT USING "###.##"; 1234.5
1,234.50
```

```
PRINT USING "#####.##"; 1234.5
1234.50,
```

^^^

You may place four carets (or circumflexes) after the digit position characters to specify exponential format. The four carets reserve space to print E+xx (or D+xx). Any decimal point position may be specified. Vectra BASIC left-justifies the significant digits and adjusts the exponent accordingly. Unless you include either a plus formatting character or a trailing plus or minus formatting character, Vectra BASIC reserves one space to the left of the decimal point to print a space (for positive numbers) or a minus sign (for negative numbers).

```
PRINT USING "###.####"; 234.56
2.35E+02
```

```
PRINT USING "#####.##"; -88888
-888E+05-
```

```
PRINT USING "#####"; 123
+.12E+03
```

An underscore character in the format string prints the next character as a literal character.

```
PRINT USING " _###.## _!"; 12.34
112.34!
```

You may include the underscore character within the formatting string by preceding it with an underscore. The next example contains a string constant within the format string.

```
PRINT USING "EXAMPLE _"; 1
EXAMPLE _1
```

Vectra BASIC prints a percent sign (Z) before a number when the printed value exceeds the specified numeric field. When rounding causes the number to exceed the field length, Vectra BASIC prints the percent sign before the rounded number.

```
PRINT USING "###.##"; 111.22  
Z111.22
```

```
PRINT USING "###.##"; .999  
Z1.00
```

If the number of digits exceeds 24, an illegal function call results.

## PRINT # and PRINT# USING Statements

**Format:** PRINT# *filename*, [USING *stringexp1*] *list of expressions*

**Purpose:** Writes data to a sequential disc file or device. You normally use these statements when writing reports. PRINT# USING is especially helpful where the appearance of the document is critical.

**Remarks:** *filename* is the number you gave the file when you opened it for output.

*stringexp* consists of the formatting characters as described for the PRINT USING statement.

The expressions in *list of expressions* are the numeric and/or string values that you want to write to the file.

PRINT# does not compress data on the disc. With this statement, Vectra BASIC writes an image of the data to disc, just as it would display the information on your computer screen. For this reason, it is useful when writing reports, but not for storing ASCII data to disc.

For example, let A\$ = "CAMERA" and B% = "93604-1".

The statement:

```
PRINT #1, A$;B%
```

writes the following data to the disc:

```
CAMERA93604-1
```

Since the `PRINT` statement omitted explicit delimiters, you would be unable to use an `INPUT` statement to read both strings back in. To correct this problem, you must insert explicit delimiters into the `PRINT` statement as follows:

```
PRINT #1, A$;" ";B$
```

This statement writes the following image to disc:

```
CAMERA, 93604 - 1
```

In this form, you may use the `INPUT` statement to read both values.

When the strings themselves contain commas, semicolons, significant leading spaces, carriage return, or line feed characters, you must surround the string with explicit quotation marks, that is `CHR$(34)`.

For example, let `A$ = "CAMERA, AUTOMATIC"` and `B$ = " 93604 - 1"`.

The statement:

```
PRINT #1, A$;B$
```

writes the following image to disc:

```
CAMERA,AUTOMATIC 93604 - 1
```

Therefore, the following `INPUT` statement:

```
INPUT #1, A$,B$
```

assigns "CAMERA" to `A$` and "AUTOMATIC 93604 - 1" to `B$`.

To separate these strings properly on the disc, you must include double quotes within the string by using `CHR$(34)`.

The statement:

```
PRINT #1, CHR$(34);A$;CHR$(34);" ";CHR$(34);B$;CHR$(34)
```

writes the following image to disc:

```
"CAMERA, AUTOMATIC"," 93604 - 1"
```

Therefore, the statement:

```
INPUT #1, A$,B$
```

assigns "CAMERA, AUTOMATIC" to `A$` and " 93604 - 1" to `B$`.

When you use the `WRITE` statement, Vectra BASIC automatically includes all delimiters and quotation marks for you. Therefore, it is more appropriate to use `WRITE` for storing data to disc.

## PSET Statement

**Format:** PSET (STEP) (x,y) (color)

**Purpose:** Draws a pixel at the specified coordinates with the given attribute.

**Remarks:** This command is valid only in a graphics mode.

x and y are the coordinates of the point you wish to set.

In medium resolution mode, x can equal 0 to 319; in high resolution, x can equal 0 to 639. y can range from 0 to 199 in both graphics modes.

color specifies which color to use. In medium resolution, 0 selects the background color. 1-3 select colors from the palette chosen by the COLOR statement. In high resolution, 0 selects the background color and 1 selects the foreground color as set by a COLOR statement.

When you omit color, Vectra BASIC uses the foreground color. In medium resolution, the default foreground color is 3.

PRESET works exactly like PSET, except that PRESET uses the background color if color is not specified.

You may give the x and y coordinates in absolute or relative form. When using the relative form, you must give the offset from the most recently referenced point with the STEP option:

STEP (xoff/set, yoff/set)

When Vectra BASIC scans coordinate values, it ignores values that lie beyond the edge of the screen. However, values outside the integer range -32768 to 32767 cause an *Overflow* error.

### Example:

This example draws a line from (0,0) to (100,100) and then erases that line by overwriting it with the background color:

```
5 REM Draw a line from (0,0) to (100,100)
6 SCREEN 1
10 FOR COORD = 0 TO 100
20   PSET (COORD, COORD), 2
30 NEXT
35 REM Now erase that line
40 FOR COORD = 0 TO 100
50   PSET STEP (-1,-1), 0
60 NEXT
```

## PUT Statement

**Format:**

PUT ( # ) *filenum* ( , *record* )

**Purpose:**

Writes a record from the random file buffer to a random-access disc file.

**Remarks:**

*filenum* is the number you gave the file when you opened it.

*record* identifies the record to be written. It may range from 1 to 32767.

When you omit *record*, Vectra BASIC uses the next available record number (after the last PUT).

**Note**

You may use PRINT, PRINT USING, and WRITE to put characters in the random file buffer before a PUT statement executes. When you use the WRITE statement, Vectra BASIC pads the buffer with spaces up to the carriage return character. Attempting to read or write beyond the end of the buffer causes a `field overflow error`.

**Example:**

```
10 OPEN "R", #1, "BDGT", 30
20 FIELD #1, 18 AS PAYEE$, 4 AS AMT$, 8 AS DATE$
30 INPUT "ENTER CHECK NUMBER"; CK%
40 INPUT "PAYEE"; PAY$
50 INPUT "DOLLAR AMOUNT"; A
60 INPUT "DATE"; D$
70 LSET PAYEE$ = PAY$
90 LSET AMT$ = MK$(A)
90 LSET DATE$ = D$
100 PUT #1, CK%
110 GOTO 30
```

## PUT Statement (for graphics applications)

In addition to the standard PUT statement, the graphics format of PUT is:

**Format:**

PUT ( *x1*, *y1* ) , *arrayname* ( , *action* )

**Purpose:**

Transfers an image from the specified array to the screen.

**Remarks:**

The GET statement saves an image in an array. PUT places the saved image on the screen. See the GET statement for more information.

*x1*, *y1* are the coordinates of the top left corner of the transferred image.

The entire image must fit within the screen boundary, or an illegal function call message results. In medium resolution, *x1* can range from 0 to 319 less the image width; in high resolution, *x1* can range from 0 to 639 less the image width. In both graphics modes, *y1* can range from 0 to 199 less the image height.

*arrayname* is the name of the numeric array that contains the data.

*action* is one of the following values:

- |               |                                                                                                                                                                                                           |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PSET</b>   | Transfers the data verbatim.                                                                                                                                                                              |
| <b>PRESET</b> | Produces a negative image. In high resolution, background and foreground are transposed. In medium resolution, values from the array are reversed: 0 becomes 3, 3 becomes 0, 1 becomes 2 and 2 becomes 1. |
| <b>AND</b>    | Transfers the image only if an image already exists under the transferred image, performing a logical AND on the bits of the images.                                                                      |
| <b>OR</b>     | Superimposes the image onto an already existing image.                                                                                                                                                    |
| <b>XOR</b>    | Inverts points on the screen that correspond to existing points in the array image. When you omit the <i>action</i> parameter, the system uses XOR as the default setting.                                |

You may use the GET and PUT statements to perform animation. This involves initiating these steps:

1. PUT the object(s) on the screen.
2. Recalculate the new position of the object(s).
3. PUT the object(s) on the screen a second time at the old location(s) to remove the old images.
4. Go to step 1, but this time PUT the object(s) at the new location.

When done in this manner, the movement leaves the background unchanged. You can decrease the amount of flicker by minimizing the time between steps 4 and 1, and by making sure a sufficient time delay exists between steps 1 and 3. If multiple objects are being animated, you should process every object at once, one step at a time.

The PAINT statement includes an example which uses PUT and GET for animation.

When it is not important to preserve the background, you may perform animation with the PSET *action* option. Here, you leave a border around the image when you GET it that is as large or larger than the maximum distance the object moves. Thus, as the object moves, the border effectively erases any extraneous points. This method may be somewhat faster than the method described above since only one PUT is needed to move the object (although the image transferred is larger).

### Example:

```

10 REM show put in different
20 REM modes against different backgrounds
30 DIM A$(164)
40 SCREEN 1:CLS
50 LINE (0,0)-(35,35),3,BF
60 LINE (5,5)-(30,30),2,BF
70 LINE (10,10)-(25,25),1,BF
80 LINE (15,15)-(20,20),0,BF
90 GET (0,0)-(35,35),A$
100 REM Black background
110 Y=40
120 GOSUB 280
130 REM Cyan background
140 Y=80
150 LINE (0,75)-(319,120),1,BF
160 GOSUB 280
170 REM Magenta background
180 Y=120
190 LINE (0-112B)-(319,160),2,BF
200 GOSUB 280
210 REM Striped background
220 Y=160
230 FOR J=Y TO 199 STEP 2:LINE (0,J)-(319,J):NEXT
240 GOSUB 280
250 LOCATE 2,6:PRINT "<-original"
260 LOCATE 5,6:PRINT "PSET" TAB(13) "PRESET" TAB(22)
    "AND" TAB(30) "OR" TAB(37) "XOR"
270 END
280 PUT (40,Y),A$,PSET
290 PUT (100,Y),A$,PRESET
300 PUT (160,Y),A$,AND
310 PUT (220,Y),A$,OR
320 PUT (280,Y),A$,XOR
330 RETURN

```

## RANDOMIZE Statement

### Format:

RANDOMIZE *Expression*

### Purpose:

Reseeds the random-number generator.

### Remarks:

When you omit *expression*, Vectra BASIC suspends program execution and asks for a value by printing:

Random number seed (-32768 to 32767)?

After you enter a value, Vectra BASIC executes the RANDOMIZE statement.

If you fail to reseed the random-number generator, the RND function returns the same sequence of "random" numbers each time you run the program. To change the seed each time the program runs, place a RANDOMIZE statement at the beginning of the program and change its argument before each run.

To obtain a new random number seed without user intervention, use the TIMER function for the *expression*, as in the following example:

```
10 RANDOMIZE TIMER
20 REM Now test the generator
30 FOR I = 1 TO 4
40 PRINT RND;
50 NEXT I
60 END
RUN
.9590051 .1036786 .1464037 .7754918
RUN
.8261163 .17422 .9791545 .4876183
```

### Example:

```
10 RANDOMIZE
20 FOR I = 1 TO 5
30 PRINT RND;
40 NEXT I
50 END
RUN
Random number seed (-32768 to 32767)?
(you type 3 [Enter])
.2226007 .5941419 .2414202 .2013798
5.361748E-02
0%
RUN
Random number seed (-32768 to 32767)?
(you type 4 [Enter])
.628988 .765605 .5551516 .775797 .7834911
0%
RUN
Random number seed (-32768 to 32767)?
(you type 3 [Enter] which produces the first sequence)
.2226007 .5941419 .2414202 .2013798
5.361748E-02
0%
```

## READ Statement

### Format:

READ *variable* [, *variable*] ...

### Purpose:

Reads values from DATA statements and assigns these values to the named variables.

### Remarks:

*variable* is a numeric or string variable that receives the value read from a DATA statement. It may be a simple variable or an array element.

You always use READ statements in conjunction with DATA statements. READ statements assign DATA items to variables on a one-to-one basis. The READ-statement variables may be numeric or string. The values in the DATA statement must agree, however, with the specified variable types. If they differ, a **Syntax error** occurs.

A single READ statement may access one or multiple DATA statements, or several READ statements may access the same DATA statement. If the number of variables exceeds the number of elements in the DATA statement(s), Vectra BASIC prints an Out of DATA error message. If the number of variables is less than the number of elements in the DATA statement, subsequent READ statements begin reading data at the point where the last READ operation finished. When no subsequent READ statements occur, Vectra BASIC ignores the extra data.

You may reread DATA statements by using the RESTORE statement. (See the RESTORE statement for more information.)

### Examples:

This example reads the values from the DATA statements into the array A. After the FOR loop, the value of A(1) is 3.08, A(2) is 5.19, and so on:

```
90 FOR I = 1 TO 10
100 READ A(I)
110 NEXT I
120 DATA 3.08,5.19,3.12,3.98,4.24
130 DATA 5.08,5.55,4.00,3.16,3.37
140 FOR I = 1 TO 10
150 PRINT A(I)
```

The following program segment reads both string and numeric data:

```
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$, S$, Z
30 DATA "DENVER", "COLORADO", 80211
40 PRINT C$, S$, Z
50 END
RUN
CITY STATE ZIP
DENVER, COLORADO 80211
```

Note that you may omit placing quotation marks around the string COLORADO since it contains no commas, semicolons, or significant spaces. However, you must place quotation marks around DENVER, because of the comma.

This program reads string and numeric data from two consecutive DATA statements until all variables have been assigned a value. The excess data is ignored:

```
10 FOR K = 1 TO 5
20 READ A$: PRINT A$
30 NEXT K
40 DATA "TONI", "NICO"
50 DATA "BOB", "BERNADETTE", S2, S0, PRINGLE
60 END
RUN
TONI,NICO,BDB,BERNADETTE$2
```

## REM Statement

### Format:

REM *remark*

### Purpose:

Inserts explanatory remarks into a program without affecting program execution.

### Remarks:

*remark* may be any sequence of characters.

When you list a program, Vectra BASIC prints REM statements exactly as you entered them. REM statements are never executed.

You may branch to a REM statement from a GOTO or GOSUB statement. In this case, execution continues with the first executable statement after the REM statement.

You may append remarks at the end of a program line by preceding the remark with a single quotation mark or apostrophe (') instead of :REM. However, you must avoid using this method at the end of a DATA statement. In this event, Vectra BASIC would interpret the remark as part of the data.

### Note

Never append programming statements to a REM line since Vectra BASIC will interpret the statements as part of the remark. For example, the following statements do not print a blank line:

```
500 REM Begin New Section : PRINT
```

Rather, make the REM statement the last statement in the line:

```
500 PRINT : REM Begin New Section
```

### Examples:

The first example uses the REM statement as a header for the FOR...NEXT loop:

```
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I = 1 TO 20
140   SUM = SUM + V(I)
150 NEXT I
```

The next example shows the use of the apostrophe (') for REM:

```
120 'CALCULATE AVERAGE VELOCITY
130 FOR I = 1 TO 20
140   SUM = SUM + V(I)
150 NEXT I
```

The last example attaches the comment to the end of the first statement of the FOR loop:

```
130 FOR I = 1 TO 20 'CALCULATE AVERAGE VELOCITY
140   SUM = SUM + V(I)
150 NEXT I
```

## RENUM Command

|                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Format:</b>  | RENUM ( <i>newnumber</i> ) ( , ( <i>oldnumber</i> ) ( , <i>increment</i> ) )                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Purpose:</b> | Renumbers the lines within a program.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Remarks:</b> | <p><i>newnumber</i> is the first line number in the new sequence. When you omit this parameter, Vectra BASIC sets the value to 10.</p> <p><i>oldnumber</i> is the line in the current program where renumbering begins. When you omit this parameter, Vectra BASIC begins with the first line in the program.</p> <p><i>increment</i> is the amount by which the numbering increases at each step. The default value is 10.</p> <p>RENUM also changes all references to line numbers in GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. When Vectra BASIC detects a nonexistent line number after one of these statements, the error message <code>Undefined line xxxxx in yyyyyy</code> appears. RENUM leaves the incorrect line number reference <code>xxxxx</code> as it was. However, the reference to line number <code>yyyyy</code> may have changed.</p> |

### Caution

Numeric constants following an ERL variable in a given expression may be treated as line references and thus modified by a RENUM statement. To avoid this problem, you should use statements similar to these:

```
L = ERL : PRINT L / 10
PRINT ERL / 10
```

You cannot use RENUM to change the order of program lines. For example, if a program contains three lines numbered 10, 20, and 30, attempting to change line 30 to line 15 to produce the new sequence 10, 15, 20 with the statement

```
RENUM 15,30
```

is illegal.

You cannot create line numbers greater than 65529. Attempting to do so causes an `illegal function call`.

### Examples:

The first example renumbers the entire program. The first line number is 10 and following line numbers are incremented by 10:

```
RENUM
```

The next example also renumbers the entire program. However, the first line number is 300, and subsequent lines are incremented by 50:

```
RENUM 300,50
```

The last example renumbers the lines beginning from 900 so they start at 1000 and increase by 20 at each step:

```
RENUM 1000,900,20
```

### Note

The BASIC compiler offers no support for this command.

## RESET Command/Statement

**Format:** RESET

**Purpose:** Forces disc file buffers to be written to disc, and closes all open files.

**Remarks:** RESET closes all open files on all drives and writes the directory track to every disc with open files.

All files must be closed before you remove a disc from its drive.

**Example:** 998 RESET  
999 END

## RESTORE Statement

**Format:** RESTORE (*line#*)

**Purpose:** Permits a program to reread DATA statements

**Remarks:** After a program executes a RESTORE statement, the next READ statement accesses the first item in the program's first DATA statement. If you specify *line#*, however, the next READ statement accesses the first item in the given DATA statement.

**Examples:** This program segment produces an Out of DATA error:

```
10 READ A,B,C
20 READ D,E,F
30 DATA 57,68,79
40 PRINT A;B;C;D;E;F
50 END
RUN
Out of DATA in 20
Ok
```

Adding a RESTORE statement between lines 10 and 20 assigns a value to all six variables:

```
10 READ A,B,C
15 RESTORE
20 READ D,E,F
30 DATA 57,68,79
40 PRINT A;B;C;D;E;F
50 END
RUN
57 68 79 57 68 79
Ok
```

## RESUME Statement

### Format:

```
RESUME  
RESUME 0  
RESUME NEXT  
RESUME line#
```

### Purpose:

Continues program execution after Vectra BASIC has performed an error recovery procedure.

### Remarks:

You may select between the various formats depending upon where you want execution to resume.

RESUME or  
RESUME 0

Execution resumes at the statement that caused the error.

RESUME NEXT

Execution resumes at the statement that immediately follows the one that caused the error.

RESUME *line#*

Execution resumes at *line#*.

A RESUME statement that is not in an error-handling routine causes a RESUME without error message.

### Example:

```
10 ON ERROR GOTO 900  
...  
900 IF (ERR=230) AND (ERR=90)  
    THEN INPUT "PRESS RETURN TO CONTINUE", A$  
910 RESUME 80  
...
```

### Note

If you plan to compile your program, see the BASIC compiler manual for differences between implementations.

## RETURN Statement

### Format 1:

```
RETURN
```

### Purpose:

Returns program control to the line immediately following the most recently executed GOSUB or ON...GOSUB statement.

### Remarks:

See the GOSUB and ON...GOSUB statements in this chapter for an example on the RETURN statement.

### Format 2:

```
RETURN line#
```

### Purpose:

Returns from an event-trapping routine to a certain place in a Vectra BASIC program. This is an extension to the standard RETURN statement.

### Remarks:

*line#* specifies the line where the event-trapping routine returns.

### Caution

You should exercise extreme caution when using the *line#* option. Bypassing the normal entries and exits to GOSUB, WHILE, and FOR statements can produce runtime errors.

### Note

If you plan to compile your program, check the BASIC compiler manual for differences between the interpretive and compiled version of this statement.

## RIGHT\$ Function

**Format:** RIGHT\$(X\$,I)

**Action:** Returns the rightmost I characters of string X\$. When I is greater than or equal to the number of characters in X\$, RIGHT\$ returns X\$. When I is zero, the function returns the null string (a string of zero length).

Also see the MID\$ and LEFT\$ functions.

**Example:**

```
10 A$ = "BASIC"
20 PRINT RIGHT$(A$,3)
RUN
SIC
OK
```

## RMDIR Statement

**Format:** RMDIR path

**Purpose:** Removes a directory from the specified disc.

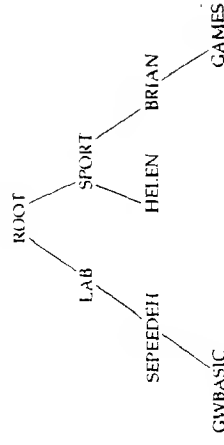
**Remarks:** path is a string expression (not exceeding 63 characters) that identifies the subdirectory.

A directory must be empty of all files and subdirectories before you can remove it.

**Examples:** The following statements delete the files from a subdirectory, then remove that directory:

```
KILL "STORIES\*.*"
RMDIR "STORIES"
```

The next examples refer to the following tree structure:



If the ROOT is the current directory, you may remove the directory called HELEN with this statement:

```
RMDIR "SPORT\HELEN"
```

You can make SEPEEDEM the current directory and delete GW BASIC with these statements:

```
CHDIR "LAB\SEPEEDEM"  
RMDIR "GW BASIC"
```

You cannot remove current directory or the current directory's parent (that is, the directory preceding the current directory). For example, with the given tree-structure directory, you cannot delete the directory SPORT if BRIAN is the current directory. Attempting to do so produces a `Path/file access error`.

Trying to use the KILL command to remove a directory also produces a `Path/file access error`.

## RND Function

**Format:** RND (X)

**Action:** Returns a random number between 0 and 1. RND generates the same sequence of "random" numbers each time a program runs unless you use the RANDOMIZE statement to reset the random-number generator. However, a negative value for X always restarts the same sequence for any given X.

Setting X to 0 repeats the last number that was generated.

Omitting X or specifying a positive X generates the next random number in the sequence.

### Example:

```
10 FOR I = 1 TO 5  
20 PRINT INT (RND * 100);  
30 NEXT  
RUN  
12 65 86 72 79  
OK
```

## RUN Command/Statement

**Format 1:**     RUN [*line#*]

**Purpose:**       Executes the program currently stored in your computer's memory.

**Remarks:**    When you include *line#*, execution begins on that line. Otherwise, execution begins with the lowest line number. Vectra BASIC always returns control to the command level when program execution finishes.

**Format 2:**     RUN *filename* [, R]

**Purpose:**       Loads a file from disc into your computer's memory and then executes it.

**Remarks:**    *filename* is the name you gave the file when you saved it. It may contain an optional drive designator and path.

If no device designator is included in *filename*, Vectra BASIC uses the current drive. If no path is specified, Vectra BASIC searches the current directory for *filename*.

Vectra BASIC will supply the filename extension .BAS if no extension is specified.

If *filename* is a literal, you must enclose the name in quotation marks.

RUN closes all open files and deletes the current contents of computer memory before loading the named program. However, when you use the R option, all data files remain open.

For further information on files, see Chapter 4.

**Examples:**    The first example executes the program currently in memory:

RUN

The next example loads the program NEWFIL from disc then runs it while keeping data files open:

RUN "NEWFIL", R

The last example uses RUN as a statement to re-execute the current program from its beginning:

9999 RUN       "Re-run program"

### Note

Differences exist between the interpretive and compiled version of the RUN command. See the BASIC compiler manual if you plan to compile your program.

## SAVE Command

### Format:

SAVE *filename* ( [, A] [, P] )

### Purpose:

Stores a program file from your computer's memory to disc.

### Remarks:

*filename* is a string expression that "names" the file for future references. It may contain an optional drive designator and path.

If no device designator is included in *filename*, Vectra BASIC uses the current drive. If no path is specified, Vectra BASIC saves the file in current directory.

Vectra BASIC will supply the filename extension .BAS if no extension is specified.

If *filename* is a literal, you must enclose the name in quotation marks.

When a file already exists on the directory with *filename*, Vectra BASIC overwrites it. No warning is given.

The A option saves the file in ASCII format. Otherwise, Vectra BASIC saves the file in a compressed binary form. ASCII format uses more disc space, but some disc accesses require that files be in ASCII format. For instance, the MERGE command requires ASCII formatted files. Also, any programs that you save in ASCII format may be read as data files.

### Note

You may also use the LIST command to write all or part of a program to a disc file in ASCII format.

The P option protects the file by saving it in an encoded binary format. When the protected file is later loaded or run, any attempt to list or edit it fails. No command exists to "unprotect" such a file.

### Examples:

The first example saves the program MYPROG in ASCII format:

```
SAVE "MYPROG", A
```

The next command saves the program STATS as a protected file that cannot be altered:

```
SAVE "STATS", P
```

The last example saves the program BDGT to the disc on drive C:

```
SAVE "C:BDGT"
```

## SCREEN Function

**Format:** SCREEN (*row*, *col* [, *z*])

**Action:** The SCREEN function returns the following:

- a. The ASCII code for a screen character when you specify that character's row and column coordinates.
- b. The character attribute if you additionally supply a *z* parameter, and this parameter is not equal to zero. (See following discussion.)

*row* is a number that ranges between 1 and 25 when the function key display is turned off. When the function key display is on, *row* can range between 1 and 24. It gives the row number.

*col* can range between 1 and 80 when the WIDTH is set to 80, or 1 and 40 when the WIDTH is set to 40. It gives the column number.

*z* is a numeric expression that yields a Boolean result. When you specify *z* and if it is not zero (that is, the Boolean result is true), the function returns the character attribute.

### Note

See the COLOR statement for a list of colors and character attributes.

In text mode, the returned value ranges from 0 to 255, and indicates the foreground color or character mode, the background color, and whether the character is blinking or not.

The following computations are needed to decipher these characteristics. In these statements, assume that the result of the function is stored in *a*.

```
Foreground color = (a MOD 16)
Background color = (((a - foreground) / 16) MOD 8)
(You must first calculate the foreground color)
```

For blinking characters, *a* will be greater than 127; for non-blinking characters, *a* will be 0-127.

In graphics modes, the SCREEN (*row*, *col*, *z*) function returns the color (set by POKE #HFE, *color*) of the character at *row*, *col*. If *row*, *col* contains graphics information, the function returns a value of 0. (It is not possible to have a character of the background color on a graphics screen.)

**Examples:** If the character at (10,20) is a capital B, then the function returns the value 66.

```
100 X = SCREEN(10,20)
```

This example prints a string of characters on the screen. It uses the SCREEN function to read the ASCII value of the string.

```
10 SCREEN 1:CLS
20 LOCATE 1,1:PRINT "This is some text"
30 FOR Y=1 TO 17
40 V=SCREEN(1,Y)
50 PRINT V,CHR$(V)
60 A$=A$+CHR$(V)
70 NEXT Y
80 PRINT A$
```

This example shows how to determine the foreground and background colors when SCREEN is used with the *z* parameter.

```
10 SCREEN 0
20 WIDTH 80
30 CLS
40 COLOR 20,0
50 LOCATE 10,10:PRINT "FLASHY"
60 A=SCREEN(10,10,1)
70 C = A MOD 16
80 B = (14-C)/16 MOD 8
90 COLOR 2
100 PRINT "A ";A; " Foreground color = ";C;"
    Background color = ";B
110 IF A>127 THEN PRINT "Blinking"
120 END
```

## SCREEN Statement

**Format:** SCREEN *mode* [*colorburst*] [*active.page*] [*visual.page*]

**Purpose:** Sets text mode, medium resolution graphics mode, or high resolution graphics mode. It can also turn off the color burst, and provide alternative screen pages in text mode.

**Remarks:** *mode* is a numeric expression that returns an integer value of 0, 1 or 2. The meanings for *mode* are:

- 0 text mode using the current screen width (40 or 80 characters). See the WIDTH statement.
- 1 medium graphics mode (320 pixels (or dots) by 200 pixels), with 40-character text width.
- 2 high resolution graphics mode (640 pixels by 200 pixels), with 80-character text width.

*colorburst* is a numeric expression that returns true or false. It enables or disables color, depending on your monitor type and the screen mode.

*active.page* is an integer expression that selects the page that PRINT and other screen output statements will write to. This parameter is valid only in text mode.

*visual.page* is an integer expression that selects the page to be displayed. If this parameter is omitted, visual page defaults to active page. This parameter is valid only in text mode.

The SCREEN statement must have at least one parameter, but you can omit any of the parameters in a SCREEN statement. The old value is assumed for omitted parameters, except for *visual.page*. When *visual.page* is omitted, it is set to *active.page*.

If the statement `SCREEN 0,1,3,0` is followed by `SCREEN , , 2`, the second statement retains *mode* 0, *colorburst* 1, and sets the active and visual pages to 2.

Specifying values outside the ranges for any of the parameters results in an `Illegal function call` error. In this event, all previous values are retained.

When all the `SCREEN` parameters are valid, and the specified `SCREEN mode` or *colorburst* is different than the current screen parameters, the following events occur:

- the new screen mode is stored
- the screen is erased
- the foreground color is set to white
- the background and border color are set to black (see the `COLOR` statement)

If *mode* and *colorburst* in a new `SCREEN` statement match the *mode* and *colorburst* from the previous `SCREEN` statement, no changes occur. The screen is not cleared.

When a `SCREEN 0 . . .` statement causes a change from a graphics mode to the text screen, the width for the text screen remains the width of the previous graphics screen. `SCREEN 1` followed by `SCREEN 0` produces 40 columns on the text screen; `SCREEN 2` followed by `SCREEN 0` produces 80 columns on the text screen. This can be changed by following the `SCREEN` statement with a `WIDTH` statement.

### Color burst effects

On a composite monitor, the color burst can be on or off. Turning the color burst off removes the “artifacts” that can make white text on the screen difficult to read on a color monitor. Artifacts result from combinations of bit patterns which combine to produce colors not normally displayed. (Microsoft’s Flight Simulator uses artifacts to produce greens, browns and blues, for example.)

These are the effects of *colorburst* on a composite monitor:

| Screen mode       | <i>colorburst</i> | Effect                   |
|-------------------|-------------------|--------------------------|
| Text mode         | 0                 | characters white         |
| Medium resolution | non-zero          | colors may be artifacted |
| High resolution   | 0                 | normal color operation   |
|                   | —                 | no effect                |

On an RGB monitor, the color burst is not actually turned off. On the medium resolution graphics screen, a *colorburst* of 0 produces palette 1, with red replacing magenta. The resulting palette consists of cyan, red, and white. A `COLOR` statement selecting palette 0 has no effect on RGB monitor with the color burst off.

### Page flipping

In text mode, you can display one text page, while writing information onto “invisible” pages. Then, by using a second `SCREEN` statement, you can “flip” to an invisible page. The *active page* and *visual page* parameters control this effect.

The *visual page* is the page that is displayed. The *active page* is the page where all screen output statements write. By default, the active and visual pages are page 0.

The number of text pages varies, depending on the WIDTH setting. At WIDTH 40, there are 8 text pages, numbered 0-7. With a setting of WIDTH 80, there are 4 text pages, numbered 0-3.

### Note

If you are using active and visual page flipping, it is possible to exit a program with the visual page and the active page set to different values. This gives the effect of locking up the machine since all screen output is being sent to the active page, but not the visual page. You must enter SCREEN 0,0,0 to restore the active and visual pages to 0. (This is the default setting for function key F10.)

### Note

Only one cursor is shared among all the pages. If you are switching active pages back and forth, you can save the cursor position for the current active page with PDST(0) and CSRLIN before changing to another active page. When you return to the original page, you can restore the cursor position using the LOCATE statement.

### Examples:

This example selects text mode with color burst disabled and sets the active page and the visual page to 0. This is the default value for function key F10.

```
10 SCREEN 0,0,0
```

This example prints information on a "hidden" active page, then prints a command on the visual page. Hitting [Enter] causes the program to proceed, and the "hidden" page of information is immediately revealed.

```
10 REM clear screen 0
30 SCREEN 0,0,0,0
30 CLS
40 REM Set active page 1, visual page 0
50 SCREEN 0,0,1,0
60 REM Now print on the active page
70 PRINT "This information is being printed"
80 PRINT "on the active page. Nothing."
90 PRINT "happens on the visual page."
100 REM Now print instructions on the visual page
110 SCREEN ..0,0
120 PRINT "Hit Enter to continue"
130 INPUT A$
140 REM Now make page 1 the visual page
150 REM and the active page
160 SCREEN ..1
170 END
```

## SGN Function

**Format:** SGN(X)

**Action:** If *x* is positive, SGN returns 1.  
If *x* is equal to zero, SGN returns 0.  
If *x* is negative, SGN returns -1.

### Example:

```
10 INPUT X
20 ON SGN(X) + 2 GOTO 30, 40, 50
30 PRINT "X<0" : GOTO 60
40 PRINT "X=0" : GOTO 60
50 PRINT "X>0"
60 END
```

## SHELL Statement

**Format:** SHELL [*command string*]

**Purpose:** Exits a Vectra BASIC program or the Vectra BASIC interpreter to run a .COM, .EXE, or .BAT program, or a DOS function.

### Remarks:

*command string* is a string expression that contains the name of a program to be run. It may also contain a pathname, and arguments to be passed to the command. If *command string* is a literal, it must be enclosed in quotation marks.

A program which is run from Vectra BASIC using shell is called a "child process". Vectra BASIC (and your program) remain in memory while the child process executes. When the child process has finished running, control returns to the Vectra BASIC program, or to Vectra BASIC command level.

SHELL loads and runs a copy of COMMAND.COM. If COMMAND.COM is not on the current path on the current disc, Vectra BASIC issues a **File not found** error and control returns to Vectra BASIC.

The SHELL statement runs COMMAND.COM with the /C switch, allowing command line parameters to be passed to the child process. Any text in *command string* separated from the program name by at least one blank will be processed by COMMAND.COM as parameters to be passed to the program.

The program name in *command string* may have any extension you wish. If you don't include an extension, COMMAND looks for a .COM file, then a .EXE file, and finally for a .BAT file. (.BAT files must always end with the word EXIT.) If COMMAND cannot find the file specified in *command string*, the error message **Bad command or file name** is printed and control returns to Vectra BASIC.

If you omit *command.string*, SHELL loads `COMMAND.COM` and gives you the DOS command prompt. You can run any valid DOS command, such as `DIR`, `TYPE` or `COPY`. To return to Vectra BASIC, type `EXIT`.

Standard input and output may be redirected.

Here are some additional cautions:

If the child process needs to change any files that the Vectra BASIC program uses, these files should be closed before using SHELL.

If your program is using redirected input or output, don't use a SHELL process to modify these files.

In most cases, you should issue a `SCREEN 0` command before using SHELL, or the child process may run in your current screen mode. In particular, SHELL from the graphics screen frequently does not display the cursor in the shelled-to application.

Returning from a SHELLED child process does not clear the screen. In most cases, you will want to issue a `SCREEN` and `CLS` command after a SHELL process.

If you used the `/M`: switch when you started Vectra BASIC, the interpreter cannot compress its workspace to make room for the SHELL process to run. If the SHELL process is too large to fit in the remaining memory, the error message *Out of memory* is printed and control returns to Vectra BASIC.

### Note

You may SHELL to GW BASIC. Some versions of BASIC do not allow BASIC as a child of BASIC.

### Examples:

This example shows the use of SHELL from command level without *command.string*.

```
Ok
SHELL
Command v. 3.10 (C)Copyright Microsoft Corp
1981, 1985
```

```
ADIR MYPROG.*
MYPROG BAS 432 9-11-85 1:44a
MYPROG DAT 26 9-11-85 1:45a
2 file(s) 128880 bytes free
```

```
A> TYPE MYPROG.DAT
Smith, Mary 841-1114
A> EXIT
Ok
```

The following command could be used to SHELL a word processing program "WP.COM" and pass it the filename "myprog.doc." When the word processing program is ended (by whatever command usually returns to DOS), control returns to Vectra BASIC.

```
Ok
SHELL "WP MYPROG.DOC"
```

## SIN Function

**Format:**        `SIN(x)`

**Action:**        Returns the sine of *x*, where *x* is given in radians.

**Note**            To convert degrees to radians, multiply the angle by  $\pi/180$ , where  $\pi = 3.141593$ .

• Vectra BASIC evaluates `SIN(x)` with single-precision arithmetic.

To achieve a double-precision result, *x* must be defined as a double-precision variable and Vectra BASIC must be invoked with the /P switch, or the results of the function must be stored in a double-precision variable, as in `J# = SIN(x)`.

**Example:**

```
PRINT SIN (1.50)
               .9974951
               0k
```

## SOUND Statement

**Format:**        `SOUND freq, duration`

**Purpose:**        Plays a sound of the specified frequency through the speaker.

**Remarks:**     *freq* is the desired frequency in hertz. This must be a numeric expression returning an unsigned integer in the range of 37 to 32767.

*duration* is the duration in clock ticks. Clock ticks occur 18.2 times per second. *duration* must be an unsigned integer in the range of 0 to 65535.

In Music Foreground mode, Vectra BASIC stores 3 `SOUND` statements in the Music Buffer. When a `SOUND` statement is executed, Vectra BASIC places the sound in the Music Buffer, and proceeds to the next statement as the note is played. If the buffer contains 3 notes, and the next statement is a `SOUND` statement, Vectra BASIC pauses until a previous sound is finished and the new `SOUND` statement is loaded into the buffer. If the next statement is not a `SOUND` statement, Vectra BASIC executes it immediately while the `SOUND` statements in the buffer are played. This mode of operation can be altered with the `PLAY "MB"` statement, which places all `SOUND` and `PLAY` notes in a Background Music buffer. The second example demonstrates the effects of `PLAY "MB"`.

### Note

See the `PLAY` statement for more information.

A `SOUND` statement with a duration of 0 turns off any current `SOUND` statement that is still running. If no `SOUND` statement is running, a `SOUND` statement with a duration of zero has no effect.

A `SOUND` statement with a *freq* of 32767 produces no tone. To create a period of silence, use `SOUND 32767, duration`.

This table shows the notes produced by different values of *freq*. Notice that notes that are one octave apart have a 2:1 ratio in their frequencies, that is, middle C has a frequency of 261.63 hertz, and the C in the octave below has a frequency of 130.81 hertz.

| <i>freq</i> | Note     |
|-------------|----------|
| 130.810     | C        |
| 146.830     | D        |
| 164.810     | E        |
| 174.610     | F        |
| 196.000     | G        |
| 220.000     | A        |
| 246.940     | B        |
| 261.630     | middle C |
| 293.660     | D        |
| 329.630     | E        |
| 349.230     | F        |
| 392.000     | G        |
| 440.000     | A        |
| 493.880     | B        |
| 523.250     | C        |
| 587.330     | D        |
| 659.260     | E        |
| 698.460     | F        |
| 783.990     | G        |
| 880.000     | A        |
| 987.770     | B        |
| 1046.500    | C        |
| 1174.700    | D        |
| 1318.500    | E        |
| 1396.900    | F        |
| 1568.000    | G        |
| 1760.000    | A        |
| 1975.500    | B        |

This table shows some of the common musical tempos, and the value for *duration* needed to produce them in a SOUND statement.

| <i>duration</i> | Beats/<br>Minute | Tempo       |
|-----------------|------------------|-------------|
| 27.3-18.2       | 40-60            | Larghissimo |
| 18.2-16.55      | 60-66            | Largo       |
|                 |                  | Larghetto   |
|                 |                  | Grave       |
| 16.55-14.37     | 66-76            | Lento       |
|                 |                  | Adagio      |
| 14.37-10.11     | 76-108           | Adagietto   |
|                 |                  | Andante     |
| 10.11-9.1       | 108-120          | Andantino   |
|                 |                  | Moderato    |
| 9.1-6.5         | 120-168          | Allegretto  |
|                 |                  | Allegro     |
|                 |                  | Vivace      |
|                 |                  | Veloce      |
| 6.5-5.25        | 168-208          | Presto      |
|                 |                  | very fast   |
|                 |                  | Prestissimo |

Examples:

The following "hearing tester" program produces successively higher sounds until a key is hit on the keyboard.

```
10 PRINT "Hit any key when you can no longer  
hear the tone"  
20 FOR F = 40 TO 32767 STEP 20  
30 SOUND F,.1  
40 IF INKEY$="" THEN 60  
50 NEXT  
60 PRINT "Your upper hearing limit is  
" F " hertz"  
70 END
```

Line 10 in the following program instructs Vectra BASIC to load all SOUND and PLAY statements into the Music Background buffer. Watch the execution of the PRINT statements in the program; then delete line 10 and RUN the program again.

```
10 PLAY "MB"
20 SOUND 40,30
30 PRINT TIME$
40 SOUND 70,25:SOUND 90,15
50 SOUND 60,20:SOUND 100,15
60 PRINT TIME$
```

The following example uses values from the charts for notes and tempos. It plays the same four notes at two different tempos.

```
10 REM Largo
20 READ N
30 IF N=0 THEN 60
40 SOUND N,28.13
50 GOTO 20
60 REM Allegro
70 RESTORE
80 READ N
90 IF N=0 THEN END
100 SOUND N,9.38
110 GOTO 80
120 DATA 440,261,293,349
130 DATA 0
```

## SPACES Function

**Format:**        SPACES(*x*)

**Action:**       Returns a string of *x* spaces, where *x* may range between 0 and 255.

When necessary, Vectra BASIC rounds *x* to an integer.

Also see the SPC function and the LSET statement.

### Example:

```
10 FOR I = 1 TO 5
20    X$ = SPACES(I)
30    PRINT X$; I
40 NEXT I
50 END
RUN
1
2
3
4
5
Ok
```

## SPC Function

**Format:** SPC(*i*)

**Action:** Prints *i* blanks. You may only use the SPC statement with the PRINT or LPRINT statements.

*i* is the number of spaces to be printed. When *i* is negative, SPC prints the null string. When *i* is greater than 255, SPC prints the number of blanks equal to  $J \bmod 255$ .

SPC rounds floating point numbers to an integer value to determine the number of blanks to print.

Also see the SPACE\$ and TAB functions.

### Example:

In the following PRINT statement, Vectra BASIC assumes that a semicolon follows SPC(15):

```
PRINT "OVER" SPC(15) "THERE"  
OVER  
Ok
```

## SQR Function

**Format:** SQR(*x*)

**Action:** Returns the square root of *x*. *x* must be a positive number or zero.

### Example:

```
10 FOR X = 10 TO 25 STEP 5  
20 PRINT X, SQR(X)  
30 NEXT X  
40 END  
RUN  
10 3.162278  
15 3.872984  
20 4.472136  
25  
Ok
```

## STICK Function

**Format:** STICK (*n*)

**Action:**

Returns the *x* and *y* coordinates of the two joysticks.

*n* is a numeric expression. It must return an unsigned integer in the range 0 to 3.

The functions of these values for *n* are:

- 0 Stores the *x* and *y* coordinates for both joysticks, and returns the *x* coordinate for joystick A.
- 1 Returns the *y* coordinate of joystick A.
- 2 Returns the *x* coordinate of joystick B.
- 3 Returns the *y* coordinate of joystick B.

**Note**

STICK(1), STICK(2) and STICK(3) do not sample the joystick. You must use STICK(0) first to store all four coordinates, and then use the STICK(*n*) function you need to read the stored values.

**Example:**

This example creates an endless loop which prints the coordinates for joystick B. It demonstrates that you must store the values with STICK(0) before reading them with STICK(2) and STICK(3).

```
10 PRINT " X Y"  
20 R = STICK(0)  
30 X = STICK(2) : Y = STICK(3)  
40 PRINT X,Y  
50 GOTO 20
```

## STOP Statement

**Format:** STOP

**Purpose:**

Ends program execution and returns control to the command level.

**Remarks:**

You normally use this statement when debugging a program. However, you may use STOP statements anywhere within a program to stop execution. Upon encountering a STOP statement, Vectra BASIC prints the following message (where *nnnnn* is the line number causing the break):

```
Break in nnnnn
```

The STOP statement differs from the END statement since the STOP statement leaves all tiles open

Vectra BASIC always returns control to the command level when a STOP statement executes. You may resume execution by giving the CONT command.

**Example:**

```
10 INPUT A,B,C  
20 K = A^2 * S.3 : L = B^3 / .25  
30 STOP  
40 M = C * K + 100 : PRINT M  
RUN  
? 1.2,3 [Enter]  
Break in 30  
0:  
PRINT L [Enter]  
30.76923  
0:  
CONT [Enter]  
115.9  
0:
```

**Note**

If you plan to compile your program, see the BASIC compiler manual for differences between the interpretive and compiled version of this statement.

**STR\$ Function**

**Format:**     STR\$(x)

**Action:**     Returns a string representation of the value of x.

Also see the VAL function.

**Example:**

This example uses STR\$ to concatenate a numeric variable and a string. Notice the STR\$ retains the space that Vectra BASIC normally incorporates before a positive value. Line 40 demonstrates one way to omit that space during the concatenation.

```
10 INPUT "CODE:" , X
20 INV$="BLX-" + STR$(X)
30 PRINT INV$
40 INV$="BLX-" + MID$(STR$(X),2)
50 PRINT INV$
```

## STRIG Statement

**Format:** STRIG ON  
STRIG OFF

**Purpose:** Enables reading of the joystick triggers.

**Remarks:** After an STRIG ON statement, Vectra BASIC checks between each statement to see if a joystick trigger has been pressed. This statement must precede any attempt to read a joystick trigger press with the STRIG(n) function.

When an STRIG OFF statement is executed, Vectra BASIC stops testing for trigger presses between statements.

### Note

This statement enables or disables reading the joystick triggers. It does not affect joystick trigger trapping. See the STRIG(n) statement which enables, disables or suspends trigger event trapping.

### Example:

```
100 STRIG ON
110 V=STRIG(0)
```

## STRIG(n) Function

**Format:** STRIG(n)

**Action:** Returns the status of a specified joystick trigger.

n must be a numeric expression that returns an unsigned integer in the range of 0 to 7, designating which trigger is to be checked. The instruction can either check to see if a trigger is currently being pressed, or it can check to see if a trigger has been pressed since the last STRIG(n) function which read that trigger.

### Note

STRIG ON must be executed before STRIG(n) function calls can read trigger values. See the STRIG statement.

The values of n can be:

- |   |                                                                                                                      |
|---|----------------------------------------------------------------------------------------------------------------------|
| 0 | Returns -1 if trigger A was pressed since the last STRIG(0) statement, and returns 0 if the trigger was not pressed. |
| 1 | Returns -1 if trigger A is currently down, and returns 0 if the trigger is not currently being pressed.              |
| 2 | Returns -1 if trigger B was pressed since the last STRIG(2) statement, and returns 0 if the trigger was not pressed. |
| 3 | Returns -1 if trigger B is currently down, and returns 0 if the trigger is not currently being pressed.              |

- 4 Returns -1 if trigger A2 was pressed since the last STRIG(4) statement, and returns 0 if the trigger was not pressed.
- 5 Returns -1 if trigger A2 is currently down, and returns 0 if the trigger is not currently being pressed.
- 6 Returns -1 if trigger B2 was pressed since the last STRIG(6) statement, and returns 0 if the trigger was not pressed.
- 7 Returns -1 if trigger B2 is currently down, and returns 0 if the trigger is not currently being pressed.

**Example:** This example sets up an endless loop, which beeps each time joystick button A is pressed.

```

10 IF STRIG(0) THEN BEEP
20 GOTO 10

```

## STRIG(n) Statement

**Format:** STRIG(n) ON  
STRIG(n) OFF  
STRIG(n) STOP

**Purpose:** Enables, disables or suspends event trapping of joystick button presses.

**Remarks:** n indicates which button is to be trapped:

| n | button |
|---|--------|
| 0 | A1     |
| 2 | B1     |
| 4 | A2     |
| 6 | B2     |

STRIG(n) ON activates the trapping of joystick button n. If there is an ON STRIG(n) GOSUB line# statement, where line# is not zero, pressing joystick button n causes program control to switch to the subroutine specified by line#.

STRIG(n) OFF deactivates the trapping of joystick button n. If the button is pressed after this statement is executed, the GOSUB is not performed, and the event is not remembered.

STRING(n) STOP suspends trapping of joystick button *n*. If the button is pressed after this statement is executed, the GOSUB is not performed immediately, but the event is remembered, and the GOSUB is performed as soon as STRING(n) ON statement is executed.

See the ON STRING statement and the STRING function for more details about reading joystick buttons and trapping joystick trigger events.

**Note**

The STRING ON and STRING OFF statements, which enable and disable the reading of the joystick triggers, are distinct from those statements, which only affect trapping the triggers.

## STRING\$ Function

**Format:**     STRING\$(*i*,*j*)  
              STRING\$(*i*,15)

**Action:**     Returns a string of length *i* whose characters all have ASCII code *j* or the first character of 15.

*i* must be an integer between 0 and 255.

**Example:**

```
10 REM THE ASCII CODE FOR THE DASH SYMBOL IS 45
20 X$ = STRING$(10,45)
30 PRINT X$ "MONTHLY REPORT" X$
RUN
-----MONTHLY REPORT-----
01
```

## SWAP Statement

- Format:** SWAP *variable1*, *variable2*
- Purpose:** Exchanges the values of two variables.
- Remarks:** *variable1* and *variable2* are the identifiers for two variables or array elements.
- You may SWAP variables of any type (integer, single precision, double precision, or string) as long as both variables are of the same type. If the types for the variables differ, a `Type mismatch` error occurs.

**Example:**

```
10 A$ = " ONE" : B$ = " ALL" : C$ = " FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
ONE FOR ALL
ALL FOR ONE
OK
```

## SYSTEM Command/Statement

- Format:** SYSTEM
- Purpose:** Leaves the Vectra BASIC environment and returns control to the operating system.
- Remarks:** The SYSTEM command closes all files and reloads the MS-DOS operating system without deleting any programs or memory except Vectra BASIC and its workspace.
- You may enter this statement as a Direct Mode command or you may include it as a program statement. For example, if you called Vectra BASIC through a Batch file from MS-DOS, the SYSTEM command returns control to the Batch file. The Batch file then continues its execution from the point where it left off.
- Simultaneously pressing the `[CTRL]` and `[Break]` keys always returns you to the Vectra BASIC command level, not to the MS-DOS operating system, except when Input has been redirected.

**Note**

The BASIC compiler offers no support for this command.

## TAB Function

**Format:** TAB(*i*)

**Action:** Spaces to the *i*th position on the line. If the current print position is beyond space *i*, TAB proceeds to that position on the next line.

Values for *i* may range between 1 to 255. 1 is the leftmost position on a line; the rightmost position is the width minus one. When *i* is negative, TAB treats it as the first character position (that is, *i* = 1).

When *i* is greater than the width of the screen or printer, TAB rounds the value then calculates the value of  $J \bmod W$ , where *W* is the width of the screen or printer. TAB uses the resulting value, moving to that position on the next line. For example, if the current width were 80, TAB(95) would move to character position 15 on the next line.

You may only use the TAB statement with either the PRINT or LPRINT statements.

If TAB is the last item in a print statement, Vectra BASIC performs the TAB, but does not print a return, just as if the line had ended in a semicolon.

### Note

If SCRN: is being used as an output device, you should issue an explicit WIDTH *n* *filename*, size to insure proper TAB functioning.

### Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "MALLORY ALLISON", "$25.00"
RUN
NAME          AMOUNT
MALLORY ALLISON  $25.00
01
```

## TAN Function

**Format:** TAN(*x*)

**Action:** Returns the tangent of *x*, where *x* is given in radians.

### Note

To convert degrees to radians, multiply the angle by  $\pi/180$ , where  $\pi = 3.141593$ .

Vectra BASIC evaluates TAN(*x*) with single-precision arithmetic. If the calculation overflows, Vectra BASIC displays the Overflow error message, sets the result to machine infinity with the appropriate sign, and continues execution.

**Example:** PRINT TAN(2.22)  
-1.317612

## TIMES Function

**Format:**

TIME\$

**Action:**

Retrieves the current system time.

The TIME\$ function returns an eight-character string in the form:

hh:mm:ss

where:

hh is the hour of the day, based upon a 24-hour clock. Values range from 00 to 23.

mm is the number of minutes. Values range from 00 to 59.

ss is the number of seconds. Values range from 00 to 59.

TIME\$ is set from the MS-DOS system clock when Vectra BASIC is invoked. It can be reset by the TIME\$ statement.

**Example:**

This example assumes that the current time is 8:45 P.M.:

```
PRINT TIME$
20:45:00
```

## TIMES Statement

**Format:**

TIME\$ = string

**Purpose:**

Sets the time for subsequent use by the TIME\$ function.

**Remarks:**

string represents the current time. It may take one of the following forms:

hh Sets the hour. (Values may range from 0 to 23.) Vectra BASIC sets both minutes and seconds to 00.

hh:mm Sets both hour and minutes. (Values for minutes may range from 0 to 59.) Vectra BASIC sets seconds to 00.

hh:mm:ss Sets hour, minutes, and seconds. (Values for seconds may range from 0 to 59).

Since the computer uses a 24-hour clock, you must add 12 hours to all times after 12 noon. For example, 8:00 P.M. is 20:00. You may omit leading zeroes.

This statement resets the MS-DOS system clock.

**Example:**

```
TIME$ = "14:00"
OK
PRINT TIME$
14:00:07
OK
TIME$ = "14:34:04"
OK
PRINT TIME$
14:34:10
OK
TIME$ = "3:4:5"
OK
PRINT TIME$
03:04:10
OK
```

## TIMER Function

**Format:** `v = TIMER`

**Action:** Gives the number of seconds since midnight.

`v` is a single-precision number that represents the number of seconds since midnight.

This is a read only function. You may not use `TIMER` as a user variable.

**Example:**

```
10 REM Set time to one second before midnight
20 TIME$ = "23:59:59"
30 FOR I = 1 TO 20
40 PRINT "TIME$ = "; TIME$, "TIMER = "; TIMER
50 NEXT I
RUN
TIME$ = 23:59:59    TIMER = 86399
TIME$ = 23:59:59    TIMER = 86399
TIME$ = 00:00:00    TIMER = 0
. . .
TIME$ = 00:00:03    TIMER = 3
TIME$ = 00:00:03    TIMER = 3
TIME$ = 00:00:04    TIMER = 4
TIME$ = 00:00:04    TIMER = 4
```

## TIMER Statement

**Format:** `TIMER ON`  
`TIMER OFF`  
`TIMER STOP`

**Purpose:** Enables, disables, or suspends `TIMER` event trapping for those applications that require an interval timer.

**Remarks:** `TIMER ON` Enables `TIMER` event trapping  
`TIMER OFF` Disables `TIMER` event trapping  
`TIMER STOP` Suspends `TIMER` event trapping

You must issue a `TIMER ON` statement to activate trapping by an `ON TIMER` statement.

**Note** See the `ON TIMER` statement for further details.

`TIMER OFF` turns trapping off.

`TIMER STOP` stops trapping but the `TIMER` activity is remembered so that as soon as a `TIMER ON` statement is encountered, a trap occurs.

After the trap occurs, an automatic `TIMER STOP` statement executes. Thus, recursive traps are impossible. The `RETURN` from the trap routine automatically does a `TIMER ON` statement unless an explicit `TIMER OFF` statement executes inside the trap routine.

Event trapping only happens when Vectra BASIC is running a program. When an error trap (resulting from an `ON ERROR` statement) occurs, all trapping is automatically disabled. This includes all `ERROR`, `COM(n)`, `KEY(n)`, `PEN`, `PLAY`, and `STRIG(n)` statements.

You can use a `RETURN line #` statement in the trapping routine to return to a specific line number. However, you must exercise caution when using the `RETURN` statement in this manner. For example, any other `GOSUBs`, `WHILEs`, or `FORs` that were active when the trap occurred will remain active.

#### Example:

This example displays the time of day, every minute, on the first line of the screen.

```
10 REM Show time on line 1 every 60 seconds
20 ON TIMER(60) GOSUB 5000
30 TIMER ON
...
5000 REM Time message subroutine
5010 X = CSRLIN
5020 Y = POS(0)
5030 LOCATE 1,1 : PRINT TIME$;
5040 LOCATE X,Y : Restore old row and column
5050 RETURN
```

## TRON/TROFF Statements

### Format:

```
TRON
TROFF
```

### Purpose:

Traces the execution of program statements.

### Remarks:

You may use the `TRON` statement as a debugging aid in either Direct or Indirect Mode.

The `TRON` statement enables a trace flag. Once set, the trace prints each line number (surrounded by square brackets) when Vectra BASIC executes that line.

You can disable the trace flag by giving either a `TROFF` statement or a `NEW` command.

### Example:

```
TRON
Ok
10 K = 10
20 FOR J = 1 TO 2
30   L = K + 10
40   PRINT J;K;L
50   K = K + 10
60 NEXT J
70 END
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
RUN
1 10 20
2 20 30
Ok
```

**Note**

If you plan to compile your program, see the BASIC compiler manual for differences in the implementation of these statements.

## USR Function

**Format:**      USR (*digit*) ( *Argument* )

**Action:**      Calls an assembly-language subroutine.

*digit* specifies which USR function routine is being called. *digit* may range between 0 and 9 and corresponds to the digit you gave the function with the DEF USR statement for that routine.

When you omit *digit*, Vectra BASIC assumes USR0. See DEF USR for further details.

*Argument* is the value you are passing to the subroutine. It may be any numeric or string expression.

In this implementation, if you use a segment other than the default Data Segment (DS), you must execute a DEF SEG statement before giving a USR function call. The address given in the DEF SEG statement determines the address of the subroutine.

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed.

**Example:**

```
100 DEF SEG = 4HF000
110 DEF USR0 = 0
120 X = Y
130 Y = USR0(X)
140 PRINT Y
```

**Note**

See Appendix C. Assembly Language Subroutines.

## VAL Function

**Format:** VAL(*string*)

**Action:** Returns the numeric value for the string *string*. For example, evaluating the following function gives a result of -3:

```
VAL("-3")
```

The VAL function strips leading blanks, tabs, and line feed characters from the argument string.

### Example:

In the following program, VAL converts ZIP# to a decimal value for comparisons. Lines 20 and 30 show how you may format an IF statement by using the line feed character (Control-J).

```
10 READ FIRST$, CITY$, STATE$, ZIP$
20 IF VAL(ZIP$) < 90000 OR VAL(ZIP$) > 96699
   THEN PRINT FIRST$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$) >= 90601 AND VAL(ZIP$) < 90815
   THEN PRINT FIRST$ TAB(25) "LONG BEACH"
40 DATA MARY, CORVALLIS, OREGON, 97330
```

## VARPTR Function

**Format:** VARPTR(*variable*)  
VARPTR(*filename*)

**Action:** *filename* is the number associated with a currently opened file.

*variable* is a string expression associated with a variable.

When using the *variable* format, the command returns the address of the first byte of data identified with *variable*.

You must assign a value to *variable* before you use it as an argument to VARPTR. Failing to follow this procedure results in an **illegal function call**.

You may use a variable name of any type (numeric, string, or array).

You normally use VARPTR to obtain the address of a variable or an array so you may pass the address to an assembly-language subroutine.

When passing an array, the best procedure is to pass the lowest-addressed element of that array. Therefore, you should make the function call in the following form when accessing arrays:

```
VARPTR(A(0))
```

For string variables, VARPTR returns the first byte of the string descriptor.

### Note

You should assign all simple variables before you use VARPTR with an array argument. This is a safeguard since array addresses change whenever you assign a new simple variable.

If you use the *filename* option, **VARPTR** returns the starting address of the disc I/O buffer assigned to *filename*.

The following chart lists the values that can be read from the Vectra BASIC File Control Block. The second example demonstrates reading a File Control Block.

The structure of the File Control Block is:

| Offset | Length | Description                                                                                                                                                               |
|--------|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0      | 1      | File was opened for:<br>1-Input<br>2-Output<br>4-Random<br>16-Append                                                                                                      |
| 1      | 38     | MS-DOS file control block                                                                                                                                                 |
| 39     | 2      | Sequential files: number of sectors read or written.<br>Random files: 1 + the last record number read or written.                                                         |
| 41     | 1      | Number of bytes in the sector read or written.                                                                                                                            |
| 42     | 1      | Number of bytes left in the input buffer.                                                                                                                                 |
| 43     | 3      | Reserved.                                                                                                                                                                 |
| 46     | 1      | Device number:<br>0.1 Drives A: and B:<br>248 LPT3:<br>249 LPT2:<br>250 COM2:<br>251 COM1:<br>252 CASI: (not supported in Vectra)<br>253 LPT1:<br>254 SCRNI:<br>255 KYBD: |

| Offset | Length | Description                                                                                          |
|--------|--------|------------------------------------------------------------------------------------------------------|
| 47     | 1      | Width of the device.                                                                                 |
| 48     | 1      | For <b>PRINT#</b> , position in buffer.                                                              |
| 49     | 1      | Internal use.                                                                                        |
| 50     | 1      | Output position for use during <b>TAB</b> expansion.                                                 |
| 51     | 128    | Data buffer — used to transfer data to/from MS-DOS.                                                  |
| 179    | 2      | Record length — default = 128                                                                        |
| 181    | 2      | Current physical record number.                                                                      |
| 183    | 2      | Current logical record number.                                                                       |
| 185    | 1      | Reserved.                                                                                            |
| 186    | 2      | For disk files: position for <b>PRINT#</b> , <b>INPUT#</b> and <b>WRITE#</b> .                       |
| 188    | n      | <b>FIELD</b> data buffer size. This is set using the <b>/S</b> : option when invoking <b>BASIC</b> . |

For either format, the function returns a number that ranges between -32768 and 32767. If the value is less than 0, add 65536 to obtain the correct offset. This number is the required offset into Vectra BASIC's Data Segment (DS).

### Examples:

```
100 MYSUB = VARPTR (SUB(0))
110 CALL MYSUB
```

This program prints the values stored in the File Control Block.

```
10 OPEN "MYFILE" FOR OUTPUT AS #1
20 DEF SEG
30 FOR OFFSET = 0 TO 198
40 PRINT OFFSET, PEEK (VARPTR(#1) + OFFSET)
50 IF OFFSET>0 THEN IF OFFSET MOD 20 = 0 THEN INPUT AS:CLS
60 NEXT
```

## VARPTR\$ Function

**Format:**     `VARPTR$(variable)`

**Action:**     Returns a character form of the memory address for the given variable.

*variable* is the name of a program variable. You must assign a value to *variable* before you execute the `VARPTR$` function. Otherwise, an `Illegal function call` error occurs. You may use any variable type (numeric, string, or array).

You normally use `VARPTR$` with the `DRAW` statement in programs that you plan to compile.

`VARPTR$` returns a three-byte string in the form:

byte 0 = *type*  
          where *type* =  
                  2 indicates integer  
                  3 indicates string  
                  4 indicates single precision  
                  8 indicates double precision  
byte 1 = low byte of address  
byte 2 = high byte of address

Note, however, that the individual parts of the string are not considered characters.

The returned value is the same as:

`CHR$(type) + MK1$(VARPTR(variable))`

### Note

Because array addresses change whenever you assign a new variable name, you should always assign all simple variables before calling `VARPTR$` for an array element.

### Example:

`DRAW "X" * VARPTR$(A$)`

## VIEW Statement

### Format:

```
VIEW (SCREEN) [(vx1,vy1)-(vx2,vy2)] [,fill] [,border]
```

### Purpose:

Defines subsets of the screen, called viewports, onto which Vectra BASIC maps graphics displays. (You may only use these viewports in graphics mode.)

### Remarks:

A VIEW statement without any arguments defines the entire screen as the viewport.

SCREEN is an optional parameter. When you include this parameter, all points are plotted as absolute coordinates. Their *x* values may be inside or outside the screen limits. Only those points within the viewport limits, however, are visible. For example, if you execute the statement:

```
VIEW SCREEN (10,10)-(200,100)
```

then the point plotted by the statement `PSET (0,0),3` does not appear on the screen since 0.0 is outside of the viewport. `PSET (10,10),3` is within the viewport and plots the point in the upper-left corner.

When you omit the SCREEN parameter, all points are relative to the viewport. That is, Vectra BASIC adds the *x* and *y* values in graphics commands to the *vx* and *vy* values of the upper left corner of the viewport before plotting begins. For example, if you execute the statement:

```
VIEW (10,10)-(200,100)
```

then the statement `PSET (0,0),3` plots the point at the actual screen location 10,10.

*vx1*, *vy1* - *vx2*, *vy2* are the upper left and lower right coordinates of the viewport. They must be within the legal limits of the screen or an illegal function call occurs.

In medium resolution mode, permissible values for *vx* are 0-319; for *vy*, they are 0-199.

In high resolution mode, permissible values for *vx* are 0-639; for *vy*, they are 0-199.

*fill* is an optional parameter that permits the tiling of the view area. *fill* selects a color from the palette chosen in a COLOR statement. 0 fills the view area with the background color. If you omit this parameter, the view area is not tiled.

*border* draws a boundary line around the viewport if space is available. *border* selects a color from the palette chosen in a COLOR statement. If you omit this parameter, no border is drawn around the viewport.

VIEW sorts the argument pairs, placing the smallest values for *vx* and *vy* first. For example,

```
VIEW (100,100)-(5,5) becomes
```

```
VIEW (5,5)-(100,100)
```

The only illegal pairing of coordinates sets *vx1* = *vx2* and/or *vy1* = *vy2*. For example, the statement `VIEW (10,50)-(10,180)` produces an illegal function call error message.

All other possible pairings of *vx* and *vy* are valid. For example,

```
VIEW (20,50)-(50,20) becomes
```

```
VIEW (20,20)-(50,50)
```

You may define multiple viewports, but only one viewport is active at a time. The RUN and SCREEN commands disable all viewports.

You can scale an object by changing the size of a viewport. First, you must use the WINDOW statement to redefine the coordinates of the screen boundaries (world coordinates) to 0. After you do

this, all defined viewports have the same coordinate boundaries as the current world coordinates. Therefore, a figure that occupies half of the screen before you define the viewport, now takes up half of the viewport.

When using VIEW, the CLS (Clear Screen) statement only clears the current viewport. Using CLS with a viewport homes the graphics cursor (the "last referenced point") to the center of the viewport, but does not home the text cursor. However, you can "home" the text cursor and clear the entire screen by simultaneously pressing the **CTRL** and **Home** keys or **CTRL** and **L** in direct mode, or **PRINT CHR\$(12)** within a program. These commands also reset the last referenced point to the center of the viewport. **CLS**, **CTRL** and **L**, **CTRL** **Home** and **PRINT CHR\$(12)** all leave the viewport in effect.

### Examples:

The following example defines four viewports.

```
10 SCREEN 2:CLS
15 KEY OFF
20 LOCATE 2,2: PRINT "Viewport 1"
30 LOCATE 2,40: PRINT "Viewport 2"
40 LOCATE 14,2:PRINT "Viewport 3"
50 LOCATE 14,40:PRINT "Viewport 4"
60 VIEW (1,1)-(275,95),,1:GOSUB 1000
70 VIEW (300,1)-(575,95),,1:GOSUB 2000
80 VIEW (1,100)-(275,195),,1:GOSUB 3000
90 VIEW (300,100)-(575,195),,1:GOSUB 4000
100 END
1000 REM Draw a circle in first viewport
1010 CIRCLE (135,48),50
1020 RETURN
2000 REM Draw a box in the second viewport
2010 LINE (50,25)-(225,75),1,B
2020 RETURN
3000 REM Draw spots in the third viewport
3010 FOR D=0 TO 360 STEP 10
3020 DRAW "TA=0;NU20"
3030 NEXT
3040 RETURN
4000 REM Draw a triangle in the fourth viewport
4010 PSET (100,65),1
4020 DRAW "E38;F38;L75"
4030 RETURN
```

The next example demonstrates the scaling of an object by changing the viewport:

```
10 SCREEN 2:CLS
20 REM Define window
30 , Lower left=(0,0); upper right=(100,100)
40 WINDOW (0,0)-(100,100)
50 CIRCLE (50,50),10
60 REM Define viewport--lower left and upper
70 , right retain last world coordinates
80 VIEW (300,1)-(450,100),,7
90 CIRCLE (50,50),10
100 REM Circle is smaller, but still in center
110 END
```

This program demonstrates alternating viewports.

```
10 REM Paint a bunch of circles with
20 , alternating viewports.
30 SCREEN 2
40 FOR I = 1 TO 3
50 VIEW (10,50)-(280,120),0,1
60 VIEW (200,10)-(470,80),1,0
70 FOR J = 1 TO 50
80 VIEW SCREEN (10,50)-(280,120)
90 A = RND * 250
100 B = RND * 90 + 40
110 R = RND * 35
120 CIRCLE (A,B),R
130 PAINT (A,B)
140 VIEW SCREEN (200,10)-(470,80)
150 A = RND * 250 + 150
160 B = RND * 90
170 R = RND * 35
180 CIRCLE (A,B),R,0
190 PAINT (A,B),0
200 NEXT J
210 NEXT I
220 END
```

## VIEW PRINT Statement

|                 |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Format:</b>  | VIEW PRINT ( <i>top_line</i> TO <i>bottom_line</i> )                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Purpose:</b> | Sets the boundaries of the screen text window.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Remarks:</b> | <p><i>top_line</i> and <i>bottom_line</i> are numeric integers that indicate the top and bottom rows for a text window. They can range from 1-24 when the function keys are displayed, or from 1-25 with the function key display turned off.</p> <p>VIEW PRINT without <i>top_line</i> and <i>bottom_line</i> sets the entire screen window as the text window, restoring the normal screen boundaries.</p> <p><i>bottom_line</i> must be equal to or greater than <i>top_line</i>, or an illegal function call error is generated.</p> <p>VIEW PRINT places the cursor on the first line of the text window. All printing and scrolling takes place within the text window. You cannot use the LOCATE statement or the SCREEN function on rows which are outside the window.</p> <p>CLS clears the entire screen, and positions the cursor on the first line of the text window.</p> |

### Example:

This example prints information at the bottom of the screen, and then sets the text window above it. The FILES statement is included to provide enough information to scroll the screen.

```
5 KEY OFF
10 SCREEN 1
20 VIEW PRINT
30 COLOR 1,0
40 LOCATE 25,1:PRINT DATE$;
50 LOCATE 24,1:PRINT TIME$;
60 VIEW PRINT 1 TO 23
70 FILES
```

## WAIT Statement

|                 |                                                                                                |
|-----------------|------------------------------------------------------------------------------------------------|
| <b>Format:</b>  | WAIT <i>port</i> , <i>i</i> [, <i>j</i> ]                                                      |
| <b>Purpose:</b> | Suspends program execution while monitoring the status of a machine input port.                |
| <b>Remarks:</b> | <i>port</i> is a port number, which may range from 0 to 65535.                                 |
| <b>Note</b>     | This port is a microprocessor port; not one of your computer's datacomm (or peripheral) ports. |

*i* and *j* are integer expressions that may range from 0 to 255.

The WAIT statement suspends program execution until the specified machine input port develops a specified bit pattern. The data read at the port is XOR'ed with the integer expression *i*, and then AND'ed with *j*. When the result is zero, Vectra BASIC loops back and reads the data at the port again. When the result is not zero, execution continues with the next statement.

### Caution

You could possibly enter an infinite loop when using the WAIT statement. To avoid this situation, you must ensure that the specified value appears at the port sometime during program execution. If the program enters an infinite loop, you may exit the loop by simultaneously pressing the CTRL and Break keys.

### Example:

This example suspends program execution until port 32 receives a 1 in the second bit position:

```
100 WAIT 32, 2
```

## WHILE...WEND Statement

### Format:

```
WHILE expression
...
  (loop statements)
...
WEND
```

### Purpose:

Loops through a series of statements as long as the given condition is true.

### Remarks:

*expression* is a numeric expression which Vectra BASIC evaluates. If it is true (not zero), Vectra BASIC executes the *loop statements* until it encounters WEND. Vectra BASIC then returns to the WHILE statement and checks *expression*. If it is still true, Vectra BASIC repeats the entire process. When the expression becomes false, Vectra BASIC resumes execution with the statement that follows the WEND statement.

You may nest WHILE/WEND loops to any level. Each WEND matches the most recently encountered WHILE. An unmatched WHILE statement causes a WHILE without WEND error. An unmatched WEND statement causes a WEND without WHILE error.

If you are directing program control to a WHILE loop, you should always enter the loop through the WHILE statement.

### Example:

```
10 OPTION BASE 1
20 DIM A(10)
30 REM -----GET DATA-----
40 DATA 3,2,4,1,5,8,7,6,9,0
50 FOR I = 1 TO 10
60   READ A(I)
70   PRINT A(I);
80 NEXT I
90 REM -----BUBBLE SORT-----
100 J = 10
110 FLIPS = 1
120 WHILE FLIPS
130   FLIPS = 0
140   FOR I = 1 TO J-1
150     IF A(I) < A(I+1) THEN 170
160     SWAP A(I), A(I+1) : FLIPS = 1
170   NEXT I
180 WEND
190 PRINT
200 FOR I = 1 TO 10 : PRINT A(I); : NEXT I
RUN
3 2 4 1 5 8 7 6 9 0
0 1 2 3 4 5 6 7 8 9
OK
```

### Note

If you plan to compile your program, see the BASIC compiler manual for differences between the compiled and interpretive version of this statement.

## WIDTH Statement

### Format:

```
WIDTH size
WIDTH filenum, size
WIDTH dev, size
```

### Purpose:

Sets the line width in number of printed characters for the computer screen or a printer.

### Remarks:

*size* is a numeric expression that may range between 0 and 255. It gives the maximum number of characters that Vectra BASIC prints on a logical line. The default setting is 255 characters for the display screen and 80 characters for a line printer.

*size* changes the text mode line width. WIDTH 0 has the same effect as WIDTH 1.

A *size* setting of 255 gives an "infinite" line width. (That is, Vectra BASIC never inserts a carriage return character.) Both the POS and LPOS functions return 0 after the 255th character is printed on a line.

*filenum* is a numeric expression that may range between 1 and 5. This is the number of the file opened to a device.

*dev* is a string expression that identifies a device, such as an integral or line printer. Valid strings are LPT1, LPT2, LPT3, COM1, COM2, and SCRN.

WIDTH *size* sets the screen to 40 or 80 characters. 40 and 80 are the only 2 valid parameters; 40 is not valid for monochrome display adapters.

In high resolution mode, WIDTH 40 forces the medium resolution screen. Conversely, from the medium resolution screen, WIDTH 80 forces a change to high resolution mode.

WIDTH *filenum*, *size* immediately changes the width of the device associated with the file number. For example, the following two statements change LPT1 width to 75:

```
10 OPEN "LPT1:" FOR OUTPUT AS #1
20 WIDTH #1, 75
```

With this command option, you can change the width while the file is open.

WIDTH *dev*, *size* sets the line printer width. The default setting is 80 characters. The program stores the line size and uses this stored device width as soon as the program encounters an OPEN statement for OUTPUT for that device. The width stays in effect as long as the file remains open. If the device is already open when the WIDTH statement executes, the width remains the same; no change occurs.

This statement affects all line printer commands (such as LPRINT, LIST, and LIST *line printer*) since they do an implicit OPEN. Specifying WIDTH 255 for the line printer disables line folding. It has the effect of setting an "infinite" line width.

A *size* setting of 255 is the default for communication (COM) files. Changing the width for a communication file does not change the size of the transmit or receive buffer. It merely sends a carriage return character after every *size* characters.

### Example:

This example uses the third form of the WIDTH statement to set the printer line length to 62 characters, then uses the second form to change it to 68 characters.

```
10 WIDTH "LPT1:", 62
20 OPEN "LPT1:" FOR OUTPUT AS #1
...
90 WIDTH #1, 68
```

### Note

If you plan to compile your program, check the BASIC compiler manual for differences between the interpretive and compiled versions of this statement.

## WINDOW Statement

**Format:** WINDOW ( (SCREEN) (ax1,ay1) - (ax2,ay2) )

**Purpose:** Redefines the screen or viewport coordinates.

**Remarks:** When Vectra BASIC initializes graphics, it assigns a coordinate to each point on the screen. The upper left corner is (0,0). The lower right corner is (319,199) in medium resolution graphics and (639,199) in high resolution. The WINDOW statement allows you to redefine the coordinates of the boundaries of the screen (which redefines each point on the screen).

**Note** When you have defined a viewport (see the VIEW statement), you are redefining the boundaries of the viewport.

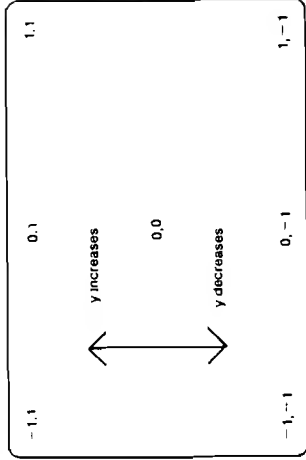
ax1,ay1 and ax2,ay2 are called "World Coordinates". They are single-precision, floating point numbers. Subsequent graphics commands such as PSET, DRAW, LINE or CIRCLE are plotted using these World Coordinates.

A WINDOW statement without any parameters sets the window to the size of the screen and returns the screen to the physical coordinate system. A RUN or SCREEN statement produces identical results.

SCREEN is an optional parameter. When you omit this parameter, the screen is viewed in true Cartesian coordinates. For example, with:

```
WINDOW (-1,-1)-(1,1)
```

the screen appears as:

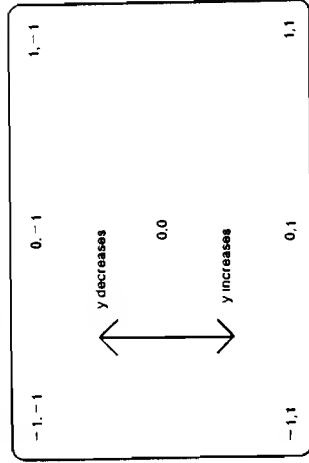


The lower left corner of the screen becomes ax1,ay1, and the upper right corner becomes ax2,ay2. This is an inversion of the usual method of defining screen space, but it is useful for certain kinds of plotting.

When you include the `SCREEN` parameter, the coordinates are not inverted so that `(ax1,ay1)` is the upper-left corner and `(ax2,ay2)` is the lower-right corner. For example, with:

```
WINDOW SCREEN (-1,-1)-(1,1)
```

the screen appears as:



`WINDOW` sorts the `ax` and `ay` argument pairs, placing the smallest values first.

For example,

```
WINDOW (100,100)-(5,5) becomes
```

```
WINDOW (5,5)-(100,100)
```

The only illegal pairing of coordinates sets `ax1=ay2` and/or `ay1=ax2`. For example, the statement `WINDOW (5,20)-(5,100)` produces an illegal function call error message.

All other possible pairings of `ax` and `ay` are valid. For example,

```
WINDOW (-5,5)-(5,-5) becomes
```

```
WINDOW (-5,-5)-(5,5)
```

Vectra BASIC clips any part of the figure that goes off the screen.

You may use the `WINDOW` statement to zoom and pan. You "zoom in" on an object by defining smaller and smaller windows while redisplaying the same object. Similarly, by defining larger and larger windows, you may "pan back" from an object.

### Example:

This example demonstrates zooming and window clipping:

```
10 SCREEN 2
20 FOR I = 10 TO 1 STEP -1
30   CLS
40   SIZE = I * 12
50   WINDOW (-SIZE,-SIZE)-(SIZE,SIZE)
60   GOSUB 1000
70   FOR J = 1 TO 500 : NEXT J 'DELAY LDDP
80   NEXT I
90 END
1000 REM DRAW A LITTLE PERSON
1010 CIRCLE (0,10),1 'HEAD
1020 LINE (-5,5)-(5,5) 'ARMS
1030 LINE (0,9)-(0,1) 'BODY
1040 LINE (0,0)-(-3,-3) 'DNE LEG
1050 LINE (0,0)-(3,-3) 'OTHER LEG
1060 RETURN
```

## WRITE Statement

**Format:** `WRITE (list of expressions)`

**Purpose:** Copies data to the computer's screen.

**Remarks:** *list of expressions* is a list of numeric and/or string expressions. You must separate the different items in the list with commas or semicolons.

When you include *list of expressions*, Vectra BASIC prints the values for the expressions on the computer screen.

• Omitting *list of expressions* prints a blank line on the screen.

When it prints the line of values, Vectra BASIC separates each item from the last with a comma. After it prints the last item in the list, Vectra BASIC inserts a carriage return/line feed. Vectra BASIC prints quotation marks around any strings within the list.

The WRITE statement prints numeric values using the same format as the PRINT statement, except that numbers are not followed by spaces, and positive numbers are not preceded by spaces.

**Example:**

```
10 X=3:Y=7:Z=7.4323456:Z$="AND THAT'S ALL"  
20 WRITE X,Y,Z$  
30 PRINT X,Y,Z$  
RUN  
3.7.432345E+08,"AND THAT'S ALL"  
3      7.432345E+08 AND THAT'S ALL  
01
```

## WRITE # Statement

**Format:** `WRITE # filename, list of expressions`

**Purpose:** Writes data to a sequential disc file.

**Remarks:** *filename* is the number you gave the file when you opened it in 0 mode.

*list of expressions* may contain numeric or string expressions or both. You must separate the items in the list with commas or semicolons.

The WRITE# statement differs from the PRINT# statement by the way it writes data to disc.

WRITE# inserts commas between the items as it writes them to disc and surrounds strings with quotation marks. Therefore, you may omit putting explicit delimiters in the list. Vectra BASIC inserts a carriage return/line feed character after it writes the last item in the list to disc.

**Example:** Let A\$ = "CAMERA" and B\$ = "93604-1" then the statement:

`WRITE #1, A$,B$`

writes the following image to disc:

`"CAMERA", "93604-1"`

A subsequent INPUT# statement, such as:

`INPUT #1, A$,B$`

assigns "CAMERA" to A\$ and "93604-1" to B\$.

## Error Codes and Error Messages

This appendix lists the Vectra BASIC error messages and describes each one.

| Number | Message |
|--------|---------|
|--------|---------|

|   |                  |
|---|------------------|
| 1 | NEXT without FOR |
|---|------------------|

A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.

|   |              |
|---|--------------|
| 2 | Syntax error |
|---|--------------|

A line is encountered that contains some incorrect sequence of characters (such as a misspelled command, unmatched parentheses, or incorrect punctuation).

|   |                      |
|---|----------------------|
| 3 | RETURN without GOSUB |
|---|----------------------|

Vectra BASIC encounters a RETURN statement for which no previous, unmatched GOSUB statement exists.

|   |             |
|---|-------------|
| 4 | Out of DATA |
|---|-------------|

Vectra BASIC is executing a READ statement but no data remains to be read from any DATA statement.

| Number | Message                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 5      | <p><b>Illegal function call</b></p> <p>You are attempting to pass a parameter that is out of the permissible range to either a string or mathematical function.</p> <p>This error message also appears under these circumstances:</p> <ol style="list-style-type: none"> <li>1. a negative or extremely large subscript</li> <li>2. a negative or zero argument to LOG</li> <li>3. a negative argument to SQR</li> <li>4. a negative mantissa with a non-integer exponent</li> <li>5. a call to an USR function for which no starting address exists.</li> <li>6. an improper argument to MID\$, LEFT\$, RIGHT\$, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or ON...GOTO</li> </ol> |
| 6      | <p><b>Overflow</b></p> <p>The result of a calculation is too large to be represented in Vectra BASIC's number format. When underflow occurs, Vectra BASIC sets the result to zero and continues execution.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 7      | <p><b>Out of memory</b></p> <p>A program is too large, has too many FOR loops or GOSUBs, has too many variables, or too many complicated expressions.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 8      | <p><b>Undefined line number</b></p> <p>A line referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement is to a nonexistent line.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

## A-2 Error Codes and Error Messages

| Number | Message                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 9      | <p><b>Subscript out of range</b></p> <p>An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.</p>                                                                                                                                                                                                                      |
| 10     | <p><b>Duplicate Definition</b></p> <p>Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.</p>                                                                                                                                                                                                   |
| 11     | <p><b>Division by zero</b></p> <p>Vectra BASIC has either encountered a division by zero within an expression or is trying to raise zero to a negative power in an exponentiation. For division by zero, Vectra BASIC sets the result to machine infinity with the sign of the numerator. For involution, Vectra BASIC sets the result to positive machine infinity. In both cases, execution continues.</p> |
| 12     | <p><b>Illegal direct</b></p> <p>You have attempted to enter a command that is illegal in Direct Mode.</p>                                                                                                                                                                                                                                                                                                    |
| 13     | <p><b>Type mismatch</b></p> <p>A string variable name is assigned a numeric value or vice versa. Otherwise, a function that expects a numeric argument is given a string argument or vice versa.</p>                                                                                                                                                                                                         |
| 14     | <p><b>Out of string space</b></p> <p>String variables have caused Vectra BASIC to exceed the amount of free memory remaining. Vectra BASIC allocates string space dynamically, until it runs out of memory.</p>                                                                                                                                                                                              |

## Error Codes and Error Messages A-3

#### Number Message

- 15 **String too long**  
An attempt is made to create a string more than 255 characters long.
- 16 **String formula too complex**  
A string expression is too long or too complex. You should break the expression into smaller expressions.
- 17 **Can't continue**  
An attempt is made to continue a program that:  
  1. has halted due to an error
  2. has been modified during a break in execution
  3. does not exist
- 18 **Undefined user function**  
A **USR** function is called before the function definition (**DEF** statement) is given.
- 19 **No RESUME**  
An error-trapping routine is entered that contains no **RESUME** statement.
- 20 **RESUME without error**  
A **RESUME** statement is encountered before an error-trapping routine is entered.
- 21 **Unprintable error**  
No error message exists for the detected error condition. This usually results from an **ERROR** statement with an undefined error code.

#### Number Message

- 22 **Missing operand**  
An expression contains an operator with no operand following it.
- 23 **Line buffer overflow**  
An attempt is made to input a line that has too many characters.
- 24 **Device timeout**  
The device you have specified is not available at this time.
- 25 **Device fault**  
An incorrect device designation has been specified.
- 26 **FOR without NEXT**  
A **FOR** was encountered without a matching **NEXT**.
- 27 **Out of paper**  
The printer device is out of paper.
- 28 **Unprintable error**  
No error message exists for the detected error condition. This usually results from an **ERROR** statement with an undefined error code.
- 29 **WHILE without WEND**  
A **WHILE** statement does not have a matching **WEND**.
- 30 **WEND without WHILE**  
A **WEND** was encountered without a matching **WHILE**.
- 31-49 **Unprintable error**  
No error message exists for the detected error condition. This usually results from an **ERROR** statement with an undefined error code.

| Number | Message                                                                                                                                                                         |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 50     | <b>Field overflow</b><br>A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.                                     |
| 51     | <b>Internal error</b><br>An internal malfunction has occurred in Vectra BASIC. Report to your Hewlett-Packard service office the conditions under which the message appeared.   |
| 52     | <b>Bad file number</b><br>A command references a file with a file number that is not opened or is beyond the range of file numbers specified at initialization.                 |
| 53     | <b>File not found</b><br>A LOAD, KILL, or OPEN statement references a file that does not exist on the current disc.                                                             |
| 54     | <b>Bad file mode</b><br>An attempt is made to use PUT, GET, or LDF with a sequential file, to LOAD a random file, or to execute an OPEN with a file mode other than I, O, or R. |
| 55     | <b>File already open</b><br>A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for an opened file.                                     |
| 56     | <b>Unprintable error</b><br>No error message exists for the detected error condition. This usually results from an ERROR statement with an undefined error code.                |

#### A-6 Error Codes and Error Messages

| Number | Message                                                                                                                                                                                   |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 57     | <b>Device I/O error</b><br>An I/O error occurred on an I/O operation. It is a fatal error since the operating system cannot recover from this error.                                      |
| 58     | <b>File already exists</b><br>The filename specified in a NAME statement is identical to a filename already in use on the disc.                                                           |
| 59-60  | <b>Unprintable error</b><br>No error message exists for the detected error condition. This usually results from an ERROR statement with an undefined error code.                          |
| 61     | <b>Disk full</b><br>All disk storage space is in use.                                                                                                                                     |
| 62     | <b>Input past end</b><br>An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. Using EOF to detect the end of file avoids this error. |
| 63     | <b>Bad record number</b><br>In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or is equal to zero.                                          |
| 64     | <b>Bad file name</b><br>An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN. (For example, the filename may contain too many characters.)                             |
| 65     | <b>Unprintable error</b><br>No error message exists for the detected error condition. This usually results from an ERROR statement with an undefined error code.                          |

#### Error Codes and Error Messages A-7

| Number | Message                                                                                                                                                                                          |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 66     | <b>Direct statement in file</b><br>A Direct Mode statement is encountered while loading an ASCII-formatted file. The LOAD is terminated.                                                         |
| 67     | <b>Too many files</b><br>An attempt is made to create a new file (using SAVE or OPEN) when all directory entries are full.                                                                       |
| 68     | <b>Device unavailable</b><br>The device that has been specified is not available at this time.                                                                                                   |
| 69     | <b>Communications buffer overflow</b><br>Not enough space has been reserved for communications I/O.                                                                                              |
| 70     | <b>Disk write protected</b><br>Your disc has a write protect tab or is a disc that cannot be written to.                                                                                         |
| 71     | <b>Disk not Ready</b><br>You have probably inserted the disc improperly.                                                                                                                         |
| 72     | <b>Disk media error</b><br>A hardware or disc problem occurred while the disc was being written to or read from. (For example, the disc drive may be malfunctioning or the disc may be damaged.) |
| 73     | <b>Advanced feature</b><br>Included for compatibility only. If a Vectra BASIC program is run under some versions of BASIC, some "advanced features" like PLAY will produce this error message.   |

| Number | Message                                                                                                                                                                                                                                        |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 74     | <b>Rename across disks</b><br>An attempt was made to rename a file with a new drive destination. As this is not allowed, the operation is canceled.                                                                                            |
| 75     | <b>Path/File Access Error</b><br>During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to make a correct Path-to-File name connection. The operation is canceled.                                                                |
| 76     | <b>Path not found</b><br>During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to find the specified path. The operation is canceled.                                                                                            |
| 77     | <b>Deadlock</b><br>Included for compatibility with future products.                                                                                                                                                                            |
| **     | <b>Can't continue after SHELL</b><br>No error number. Upon returning from a Child process, the SHELL statement discovers that there is not enough memory for Vectra BASIC to continue. Vectra BASIC closes any open files and exits to MS-DOS. |
| **     | <b>You cannot SHELL to BASIC</b><br>No error number. Included for compatibility. Vectra BASIC allows BASIC as a Child; some other versions of BASIC do not allow this procedure.                                                               |

Appendix B:

Reference Tables

- B-1 Character Sets
- B-1 Accessing the HP Character Set
- B-2 Using CHR\$
- B-3 Using the Alt Key
- B-3 ASCII Character Codes
- B-14 Scan Codes
- B-17 Reserved Words
- B-18 Syntax Charts

B

Reference Tables

Character Sets

The Vectra has two standard character sets. The text mode character set contains 256 characters. The graphics mode character set contains the first 128 characters of the text mode set.

If your Vectra is equipped with a Multi-mode card, you can use the HP character set which corresponds to the character sets on other I/P computers. This character set contains 256 characters in text mode. It cannot be used in graphics mode.

The chart on the following pages lists the Standard and HP character sets, and their ASCII equivalents.

Accessing the HP Character Set

The HP character set is only available on the Multi-mode card, and it can be used only in text mode. The statement OUT \$H3DD, n switches to this character set. The same microprocessor port controls underline mode on the text screen. These are the effects of the OUT statement:

| n | Effect                              |
|---|-------------------------------------|
| 0 | Standard set, Blue characters       |
| 2 | Standard set, Normal underline mode |
| 4 | HP set, Blue characters             |
| 6 | HP set, Normal underline mode       |

if a `COLOR` statement that results in underline mode (foreground values of 1, 9, 17 or 25) is in effect, the `OUT` statement produces blue characters for  $n=0$  and  $n=4$ , and underlined characters for  $n=2$  and  $n=6$ . If the foreground color is any other value, the `OUT` statement results in characters from the specified character set in the specified color.

Using CHRS

The statement `PRINT CHR$(n)`, where  $n$  is an ASCII value from this chart, prints the indicated character. These values of  $n$  do not produce the character, but perform the following functions:

| $n$ | Function                                |
|-----|-----------------------------------------|
| 7   | Rings the computer's "bell".            |
| 9   | Performs a horizontal tab.              |
| 10  | Line feed (with carriage return).       |
| 11  | Homes the cursor.                       |
| 12  | Clears the screen and homes the cursor. |
| 13  | Carriage return (with line feed).       |
| 28  | Cursor right.                           |
| 29  | Cursor left.                            |
| 30  | Cursor up.                              |
| 31  | Cursor down.                            |

To obtain the actual characters instead of the function, use the following `POKE` statement (text mode only):

```
DEF SEG = &HB800
POKE 2*(width*(row-1)+col-1),n
```

*width* is the screen width (40 or 80 characters), *row* and *col* are the row and column numbers, and  $n$  is the ASCII value of the character you wish to display.

Using the Alt Key

Characters which have no direct corresponding key press can be obtained by using the `CHR$` function and specifying their ASCII value, or by using the `[Alt]` key.

To obtain characters from the chart using the `[Alt]` key, hold the `[Alt]` key, then type the corresponding ASCII values for the characters on the numeric keypad. (The number keys on the alpha-numeric keyboard do NOT work.) Then, release the `[Alt]` key. If you are typing several special characters, you must release the `[Alt]` key between each character

| Value |     | Character Set |        |
|-------|-----|---------------|--------|
| Hex   | Dec | Standard      | HP     |
| 00    | 0   | Blank (Null)  | N<br>U |
| 01    | 1   | ☺             | S<br>H |
| 02    | 2   | ☹             | S<br>H |
| 03    | 3   | ♥             | €<br>H |
| 04    | 4   | ♦             | €<br>T |
| 05    | 5   | ♣             | €<br>Q |
| 06    | 6   | •             | A<br>K |
| 07    | 7   | •             | ☼      |
| 08    | 8   | ☼             | B<br>S |
| 09    | 9   | ○             | H<br>F |
| 0A    | 10  | ○             | L<br>F |
| 0B    | 11  | ♂             | V<br>T |
| 0C    | 12  | ♀             | F<br>F |
| 0D    | 13  | ♪             | C<br>R |

| Value |     | Character Set |        |
|-------|-----|---------------|--------|
| Hex   | Dec | Standard      | HP     |
| 0E    | 14  | ↵             | S<br>D |
| 0F    | 15  | ○             | S<br>I |
| 10    | 16  | ▶             | D<br>L |
| 11    | 17  | ◀             | D<br>1 |
| 12    | 18  | †             | D<br>2 |
| 13    | 19  | ''            | D<br>3 |
| 14    | 20  | qT            | D<br>4 |
| 15    | 21  | \$            | N<br>K |
| 16    | 22  | ■             | S<br>Y |
| 17    | 23  | ⏏             | E<br>B |
| 18    | 24  | †             | C<br>M |
| 19    | 25  | †             | E<br>M |
| 1A    | 26  | →             | S<br>B |
| 1B    | 27  | →             | E<br>C |
| 1C    | 28  | └             | F<br>S |
| 1D    | 29  | ↔             | G<br>S |
| 1E    | 30  | ▲             | R<br>S |

B-4 Reference Tables

| Value |     | Character Set |        |
|-------|-----|---------------|--------|
| Hex   | Dec | Standard      | HP     |
| 1F    | 31  | ▼             | V<br>S |
| 20    | 32  | Blank Space   |        |
| 21    | 33  | •             | •      |
| 22    | 34  | "             | "      |
| 23    | 35  | #             | #      |
| 24    | 36  | \$            | \$     |
| 25    | 37  | %             | %      |
| 26    | 38  | &             | &      |
| 27    | 39  | '             | '      |
| 28    | 40  | (             | (      |
| 29    | 41  | )             | )      |
| 2A    | 42  | *             | *      |
| 2B    | 43  | +             | +      |
| 2C    | 44  | ,             | ,      |
| 2D    | 45  | -             | -      |
| 2E    | 46  | .             | .      |
| 2F    | 47  | /             | /      |
| 30    | 48  | 0             | 0      |
| 31    | 49  | 1             | 1      |
| 32    | 50  | 2             | 2      |
| 33    | 51  | 3             | 3      |
| 34    | 52  | 4             | 4      |
| 35    | 53  | 5             | 5      |
| 36    | 54  | 6             | 6      |
| 37    | 55  | 7             | 7      |

Reference Tables B-5

| Value |     | Character Set |    |
|-------|-----|---------------|----|
| Hex   | Dec | Standard      | HP |
| 38    | 56  | 8             | 8  |
| 39    | 57  | 9             | 9  |
| 3A    | 58  | :             | :  |
| 3B    | 59  | ;             | ;  |
| 3C    | 60  | <             | <  |
| 3D    | 61  | =             | =  |
| 3E    | 62  | >             | >  |
| 3F    | 63  | ?             | ?  |
| 40    | 64  | @             | @  |
| 41    | 65  | A             | A  |
| 42    | 66  | B             | B  |
| 43    | 67  | C             | C  |
| 44    | 68  | D             | D  |
| 45    | 69  | E             | E  |
| 46    | 70  | F             | F  |
| 47    | 71  | G             | G  |
| 48    | 72  | H             | H  |
| 49    | 73  | I             | I  |
| 4A    | 74  | J             | J  |
| 4B    | 75  | K             | K  |
| 4C    | 76  | L             | L  |
| 4D    | 77  | M             | M  |
| 4E    | 78  | N             | N  |
| 4F    | 79  | O             | O  |
| 50    | 80  | P             | P  |
| 51    | 81  | Q             | Q  |
| 52    | 82  | R             | R  |
| 53    | 83  | S             | S  |

B-6 Reference Tables

| Value |     | Character Set |    |
|-------|-----|---------------|----|
| Hex   | Dec | Standard      | HP |
| 54    | 84  | T             | T  |
| 55    | 85  | U             | U  |
| 56    | 86  | V             | V  |
| 57    | 87  | W             | W  |
| 58    | 88  | X             | X  |
| 59    | 89  | Y             | Y  |
| 5A    | 90  | Z             | Z  |
| 5B    | 91  | [             | [  |
| 5C    | 92  | \             | \  |
| 5D    | 93  | ]             | ]  |
| 5E    | 94  | ^             | ^  |
| 5F    | 95  | _             | _  |
| 60    | 96  | `             | `  |
| 61    | 97  | a             | a  |
| 62    | 98  | b             | b  |
| 63    | 99  | c             | c  |
| 64    | 100 | d             | d  |
| 65    | 101 | e             | e  |
| 66    | 102 | f             | f  |
| 67    | 103 | g             | g  |
| 68    | 104 | h             | h  |
| 69    | 105 | i             | i  |
| 6A    | 106 | j             | j  |
| 6B    | 107 | k             | k  |
| 6C    | 108 | l             | l  |
| 6D    | 109 | m             | m  |
| 6E    | 110 | n             | n  |
| 6F    | 111 | o             | o  |

Reference Tables B-7

| Value |     | Character Set |             |
|-------|-----|---------------|-------------|
| Hex   | Dec | Standard      | HP          |
| 70    | 112 | p             | p           |
| 71    | 113 | q             | q           |
| 72    | 114 | r             | r           |
| 73    | 115 | s             | s           |
| 74    | 116 | t             | t           |
| 75    | 117 | u             | u           |
| 76    | 118 | v             | v           |
| 77    | 119 | w             | w           |
| 78    | 120 | x             | x           |
| 79    | 121 | y             | y           |
| 7A    | 122 | z             | z           |
| 7B    | 123 | {             | {           |
| 7C    | 124 |               |             |
| 7D    | 125 | }             | }           |
| 7E    | 126 | ~             | ~           |
| 7F    | 127 | Δ             | ■           |
| 80    | 128 | Ç             | HP Reserved |
| 81    | 129 | ü             | HP Reserved |
| 82    | 130 | ê             | HP Reserved |
| 83    | 131 | ä             | HP Reserved |
| 84    | 132 | ä             | HP Reserved |
| 85    | 133 | ä             | HP Reserved |
| 86    | 134 | ä             | HP Reserved |
| 87    | 135 | ç             | HP Reserved |

**B-8 Reference Tables**

| Value |     | Character Set |             |
|-------|-----|---------------|-------------|
| Hex   | Dec | Standard      | HP          |
| 88    | 136 | ê             | HP Reserved |
| 89    | 137 | e             | HP Reserved |
| 8A    | 138 | ê             | HP Reserved |
| 8B    | 139 | i             | HP Reserved |
| 8C    | 140 | ï             | HP Reserved |
| 8D    | 141 | i             | HP Reserved |
| 8E    | 142 | À             | HP Reserved |
| 8F    | 143 | À             | HP Reserved |
| 90    | 144 | É             | HP Reserved |
| 91    | 145 | æ             | HP Reserved |
| 92    | 146 | .E            | HP Reserved |
| 93    | 147 | ð             | HP Reserved |
| 94    | 148 | o             | HP Reserved |
| 95    | 149 | ö             | HP Reserved |
| 96    | 150 | ü             | HP Reserved |
| 97    | 151 | ü             | HP Reserved |
| 98    | 152 | ý             | HP Reserved |
| 99    | 153 | ö             | HP Reserved |

**Reference Tables B-9**

| Value |     | Character Set |          |
|-------|-----|---------------|----------|
| Hex   | Dec | Standard      | HP       |
| 9A    | 154 | u             | HP       |
| 9B    | 155 | €             | Reserved |
| 9C    | 156 | £             | HP       |
| 9D    | 157 | ¥             | Reserved |
| 9E    | 158 | Pl            | HP       |
| 9F    | 159 | /             | Reserved |
| A0    | 160 | •             | HP       |
| A1    | 161 | ı             | •        |
| A2    | 162 | ø             | •        |
| A3    | 163 | u             | •        |
| A4    | 164 | u             | •        |
| A5    | 165 | N             | •        |
| A6    | 166 | •             | •        |
| A7    | 167 | ø             | •        |
| A8    | 168 | •             | •        |
| A9    | 169 | •             | •        |
| AA    | 170 | •             | •        |
| AB    | 171 | •             | •        |
| AC    | 172 | •             | •        |
| AD    | 173 | ı             | •        |
| AE    | 174 | <<            | •        |
| AF    | 175 | >>            | •        |
| B0    | 176 | •             | •        |
| B1    | 177 | •             | •        |
| B2    | 178 | •             | •        |

B-10 Reference Tables

| Value |     | Character Set |    |
|-------|-----|---------------|----|
| Hex   | Dec | Standard      | HP |
| B3    | 179 | ı             | •  |
| B4    | 180 | •             | •  |
| B5    | 181 | •             | •  |
| B6    | 182 | •             | •  |
| B7    | 183 | •             | •  |
| B8    | 184 | •             | •  |
| B9    | 185 | •             | •  |
| BA    | 186 | •             | •  |
| BB    | 187 | •             | •  |
| BC    | 188 | •             | •  |
| BD    | 189 | •             | •  |
| BE    | 190 | •             | •  |
| BF    | 191 | •             | •  |
| C0    | 192 | •             | •  |
| C1    | 193 | •             | •  |
| C2    | 194 | •             | •  |
| C3    | 195 | •             | •  |
| C4    | 196 | •             | •  |
| C5    | 197 | •             | •  |
| C6    | 198 | •             | •  |
| C7    | 199 | •             | •  |
| C8    | 200 | •             | •  |
| C9    | 201 | •             | •  |
| CA    | 202 | •             | •  |
| CB    | 203 | •             | •  |
| CC    | 204 | •             | •  |
| CD    | 205 | •             | •  |

Reference Tables B-11

| Value |     | Character Set |    |
|-------|-----|---------------|----|
| Hex   | Dec | Standard      | HP |
| CE    | 206 | ⌘             | ␣  |
| CF    | 207 | ⌘             | ␣  |
| CD    | 208 | ⌘             | ␣  |
| DE    | 209 | ⌘             | ␣  |
| DF    | 210 | ⌘             | ␣  |
| EE    | 211 | ⌘             | ␣  |
| EF    | 212 | ⌘             | ␣  |
| F0    | 213 | ⌘             | ␣  |
| F1    | 214 | ⌘             | ␣  |
| F2    | 215 | ⌘             | ␣  |
| F3    | 216 | ⌘             | ␣  |
| F4    | 217 | ⌘             | ␣  |
| F5    | 218 | ⌘             | ␣  |
| F6    | 219 | ⌘             | ␣  |
| F7    | 220 | ⌘             | ␣  |
| F8    | 221 | ⌘             | ␣  |
| F9    | 222 | ⌘             | ␣  |
| FA    | 223 | ⌘             | ␣  |
| FB    | 224 | ⌘             | ␣  |
| FC    | 225 | ⌘             | ␣  |
| FD    | 226 | ⌘             | ␣  |
| FE    | 227 | ⌘             | ␣  |
| FF    | 228 | ⌘             | ␣  |
| 00    | 229 | ⌘             | ␣  |
| 01    | 230 | ⌘             | ␣  |
| 02    | 231 | ⌘             | ␣  |
| 03    | 232 | ⌘             | ␣  |

B-12 Reference Tables

| Value |     | Character Set |    |
|-------|-----|---------------|----|
| Hex   | Dec | Standard      | HP |
| E9    | 233 | ⌘             | ␣  |
| EA    | 234 | ⌘             | ␣  |
| EB    | 235 | ⌘             | ␣  |
| EC    | 236 | ⌘             | ␣  |
| ED    | 237 | ⌘             | ␣  |
| EE    | 238 | ⌘             | ␣  |
| EF    | 239 | ⌘             | ␣  |
| F0    | 240 | ⌘             | ␣  |
| F1    | 241 | ⌘             | ␣  |
| F2    | 242 | ⌘             | ␣  |
| F3    | 243 | ⌘             | ␣  |
| F4    | 244 | ⌘             | ␣  |
| F5    | 245 | ⌘             | ␣  |
| F6    | 246 | ⌘             | ␣  |
| F7    | 247 | ⌘             | ␣  |
| F8    | 248 | ⌘             | ␣  |
| F9    | 249 | ⌘             | ␣  |
| FA    | 250 | ⌘             | ␣  |
| FB    | 251 | ⌘             | ␣  |
| FC    | 252 | ⌘             | ␣  |
| FD    | 253 | ⌘             | ␣  |
| FE    | 254 | ⌘             | ␣  |
| FF    | 255 | ⌘             | ␣  |

\*Press and hold the shift key, while pressing the appropriate key  
 #To type uppercase letters, press and hold the shift key while pressing a letter key, or press  
 Caps Lock key then press the appropriate key

## Scan codes

Scan codes are used to create user-defined key traps. See the `KEY` statement.

| Scan Code |     | Key       |
|-----------|-----|-----------|
| Decimal   | Hex |           |
| 01        | 01  | ESC       |
| 02        | 02  | 1 or !    |
| 03        | 03  | 2 or @    |
| 04        | 04  | 3 or #    |
| 05        | 05  | 4 or \$   |
| 06        | 06  | 5 or %    |
| 07        | 07  | 6 or ^    |
| 08        | 08  | 7 or &    |
| 09        | 09  | 8 or *    |
| 10        | 0A  | 9 or (    |
| 11        | 0B  | 0 or )    |
| 12        | 0C  | - or _    |
| 13        | 0D  | = or +    |
| 14        | 0E  | backspace |
| 15        | 0F  | Tab       |
| 16        | 10  | Q         |
| 17        | 11  | W         |
| 18        | 12  | E         |
| 19        | 13  | R         |
| 20        | 14  | T         |
| 21        | 15  | Y         |
| 22        | 16  | U         |
| 23        | 17  | I         |
| 24        | 18  | O         |
| 25        | 19  | P         |
| 26        | 1A  | [ or {    |
| 27        | 1B  | ] or }    |
| 28        | 1C  | Enter     |
| 29        | 1D  | CTRL      |
| 30        | 1E  | A         |
| 31        | 1F  | S         |
| 32        | 20  | D         |
| 33        | 21  | F         |
| 34        | 22  | G         |
| 35        | 23  | H         |
| 36        | 24  | J         |

| Scan Code |     | Key           |
|-----------|-----|---------------|
| Decimal   | Hex |               |
| 37        | 25  | K             |
| 38        | 26  | L             |
| 39        | 27  | ; or :        |
| 40        | 28  | ' or "        |
| 41        | 29  | ` or ~        |
| 42        | 2A  | Left shift    |
| 43        | 2B  | \ or          |
| 44        | 2C  | Z             |
| 45        | 2D  | X             |
| 46        | 2E  | C             |
| 47        | 2F  | V             |
| 48        | 30  | B             |
| 49        | 31  | N             |
| 50        | 32  | M             |
| 51        | 33  | , or <        |
| 52        | 34  | . or >        |
| 53        | 35  | / or ?        |
| 54        | 36  | Right shift   |
| 55        | 37  | * or Prt Sc   |
| 56        | 38  | Alt           |
| 57        | 39  | Space bar     |
| 58        | 3A  | Caps lock     |
| 59        | 3B  | F1            |
| 60        | 3C  | F2            |
| 61        | 3D  | F3            |
| 62        | 3E  | F4            |
| 63        | 3F  | F5            |
| 64        | 40  | F6            |
| 65        | 41  | F7            |
| 66        | 42  | F8            |
| 67        | 43  | F9            |
| 68        | 44  | F10           |
| 69        | 45  | Num lock      |
| 70        | 46  | Scroll lock   |
| 71        | 47  | 7 or Home     |
| 72        | 48  | 8 or Up Arrow |
| 73        | 49  | 9 or Pg Up    |

| Scan Code |     | Key              |
|-----------|-----|------------------|
| Decimal   | Hex |                  |
| 74        | 4A  | .                |
| 75        | 4B  | 4 or Left Arrow  |
| 76        | 4C  | 5                |
| 77        | 4D  | 6 or Right Arrow |
| 78        | 4E  | +                |
| 79        | 4F  | 1 or End         |
| 80        | 50  | 2 or Down Arrow  |
| 81        | 51  | 3 or Pg Dn       |
| 82        | 52  | 0 or Ins         |
| 83        | 53  | . or DEL         |
| 84        | 54  | Sys req          |

## Reserved Words

The following table lists all the reserved words in Vectra BASIC.

|           |          |           |          |
|-----------|----------|-----------|----------|
| ABS       | ERASE    | LPRINT    | RMDIR    |
| AND       | ERDEV\$  | LSET      | RND      |
| ASC       | ERR      | MERGE     | RSET     |
| ATN       | ERL      | MID\$     | RUN      |
| AUTO      | ERR      | MKDIR     | SAVE     |
| BEEP      | ERROR    | MKD\$     | SCREEN   |
| BLOAD     | EXP      | MKI\$     | SGH      |
| BSAVE     | FIELD    | MKS\$     | SNELL    |
| CALL      | FILES    | MOD       | SIN      |
| CDBL      | FIX      | MOTOR     | SOUND    |
| CHAIN     | FNxxxxxx | NAME      | SPACE\$  |
| CHDIR     | FOR      | NEW       | SPC      |
| CHR\$     | FRE      | NEXT      | SQR      |
| CINT      | GET      | NOT       | STEP     |
| CIRCLE    | GOSUB    | OCT\$     | STICK    |
| CLEAR     | GOTO     | OFF       | STOP     |
| CLOSE     | NEX\$    | ON        | STR\$    |
| CLS       | IF       | OPEN      | STRIG    |
| COLOR     | IMP      | OPTION    | STRING\$ |
| COM       | INKEY\$  | OR        | SWAP     |
| COMMON    | INP      | OUT       | SYSTEM   |
| CONT      | INPUT    | PAINT     | TAB      |
| COS       | INPUT#   | PALETTE   | TAN      |
| CSNG      | INPUT\$  | PEEK      | THEN     |
| CSRLIN    | INSTR    | PEN       | TIME\$   |
| CVD       | INT      | PLAY      | TIMER    |
| CVI       | IOCTL    | PMAP      | TO       |
| CVS       | IOCTL\$  | POINT     | TROFF    |
| DATA      | KEY      | POKE      | TRON     |
| DATE\$    | KEY\$    | POS       | UNLOCK   |
| DEF       | KILL     | PRESET    | USING    |
| DEFDBL    | LCOPY    | PRINT     | USR      |
| DEFINT    | LEFT\$   | PRINT#    | VAL      |
| DEF SNG   | LEN      | PSET      | VARPTR\$ |
| DEFSTR    | LET      | PUT       | VIEW     |
| DELETE    | LINE     | RANDOMIZE | VIEW     |
| DIM       | LIST     | READ      | WAIT     |
| DRAW      | LLIST    | REM       | WEND     |
| EDIT      | LOAD     | RENUM     | WHILE    |
| ELSE      | LOC      | RESET     | WIDTH    |
| END       | LOCATE   | RESTORE   | WINDOW   |
| ENVIRON\$ | LOCK     | RESUME    | WRITE    |
| ENVIRON\$ | LOF      | RETURN    | WRITE#   |
| EOF       | LOG      | RIGHT\$   | XOR      |
| EOV       | LPOS     |           |          |

## COMMANDS

| Command | Syntax                                                         |
|---------|----------------------------------------------------------------|
| AUTO    | [<line number>] [<increment>]                                  |
| CONT    |                                                                |
| DELETE  | [<first line>] [-] [<last line>]                               |
| EDIT    | <line number>                                                  |
| LIST    | [<line number>] [-] [<line number>]<br>[<device>] [<filename>] |
| LLIST   | [<line number>] [-] [<line number>]<br>[<device>]              |
| MERGE   | <filename>                                                     |
| NEW     |                                                                |
| RENUM   | [<new number>] [(old number)]<br>[<increment>]                 |
| RUN     | [<line number>]                                                |
| SAVE    | <filename> [A/P]                                               |
| TROFF   |                                                                |
| TRON    |                                                                |

## STATEMENTS

| Statement | Syntax                                                                |
|-----------|-----------------------------------------------------------------------|
| BELL      |                                                                       |
| BLOAD     | <filename> [<offset>]                                                 |
| BSAVE     | <filename> , <offset> , <length>                                      |
| CALL      | <variable name> [<argument list>]                                     |
| CALLS     | <variable name> [<argument list>]                                     |
| CHAIN     | {MERGE} <filename> [<line number exp>]<br>{ALL} [DELETE <range>]      |
| CHDIR     | <path>                                                                |
| -CIRCLE   | [STEP] [<x> <y>] , <r> [<color><br>[<start> <end>] [<aspect>]]        |
| CHAIN     | {MERGE} <filename> [<line number exp>]<br>{ALL} [DELETE <range>]      |
| CLEAR     | [<max data seg>] [<max stack space>]                                  |
| CLOSE     | [#] [<file number>] [<file number>] :                                 |
| CLS       |                                                                       |
| COLOR     | [<foreground>] [( <background><br>[<border> ]]                        |
| COLOR     | [<color>] [, <palette>]                                               |
| COMI <n>  | ON<br>OFF<br>STOP                                                     |
| COMMON    | <list of variables>                                                   |
| DATA      | <list of constants>                                                   |
| DATES     | = <string expression>                                                 |
| DEF FN    | <name> [( <parameter list> )] = <function<br>definition>              |
| DEF       | <type> <range(s) of letters> where<br><type> is INT, SNG, DBL, or STR |
| DEF SEG   | [ = <address> ]                                                       |
| DEF USR   | [<digit>] = <integer expression>                                      |
| DIM       | <list of subscripted variables>                                       |
| DRAW      | <string>                                                              |



| Command     | Syntax                                                                         |
|-------------|--------------------------------------------------------------------------------|
| PAINT       | {< x >,< y > } [{< fill > } ] [{< boundary > } ]<br>[< background > ]          |
| PEN         | ON<br>OFF<br>STOP                                                              |
| PLAY        | ON<br>OFF<br>STOP                                                              |
| PLAY        | < string expression >                                                          |
| POKE        | [ ]                                                                            |
| PRESET      | [STEP] {< x coord >,< y coord > } [{< color > } ]                              |
| PRINT       | {< list of expressions > }                                                     |
| PRINT USING | < string exp >,< list of expressions >                                         |
| PRINT#      | < file number > { [ USING < string expression > ]<br>< list of expressions > } |
| PSET        | [STEP] {< x coord >,< y coord > } [{< color > } ]                              |
| PUT         | [#] {< file number > } [{< record number > } ]                                 |
| PUT         | {< x >,< y > } {>}, < array name > {< action > }                               |
| PUT         | [#] {< file number >,< number of bytes >                                       |
| RANDOMIZE   | {< expression > }                                                              |
| READ        | < list of variables >                                                          |
| REM (or )   | {< remark > }                                                                  |
| RESET       |                                                                                |
| RESTORE     | {< line number > }                                                             |
| RESUME      | [0]                                                                            |
| RESUME      | NEXT                                                                           |
| RESUME      | < line number >                                                                |
| RETURN      | {< line number > }                                                             |
| RMDIR       | < path >                                                                       |
| RSET        | < string variable > = < string expression >                                    |
| RUN         | < filename > [R]                                                               |

| Command      | Syntax                                                                                     |
|--------------|--------------------------------------------------------------------------------------------|
| SCREEN       | < mode > [ {< color burst > } ] [{< active page > } ]<br>[< visual page > ]                |
| SHELL        | [< command string > ]                                                                      |
| SOUND        | < frequency >,< duration >                                                                 |
| STOP         |                                                                                            |
| STRIG        | ON<br>OFF                                                                                  |
| STRIG(< n >) | ON<br>OFF<br>STOP                                                                          |
| SWAP         | < variable >,< variable >                                                                  |
| SYSTEM       |                                                                                            |
| TIMES        | = < string expression >                                                                    |
| TIMER        | ON<br>OFF<br>STOP                                                                          |
| VIEW         | [SCREEN] [ {< vx >,< vy > } ]<br>- [ {< vx2 >,< vy2 > } ] [ {< fill > } ]<br>[< border > ] |
| VIEW PRINT   | [< top line > TO < bottom line > ]                                                         |
| WAIT         | < port >,< i > [< n > ]                                                                    |
| WHILE/WEND   | < expression > {< loop statements > }<br>WEND                                              |
| WIDTH        | < integer expression >                                                                     |
| WIDTH        | #< file number >,< size >                                                                  |
| WIDTH        | < device >,< size >                                                                        |
| WINDOW       | [SCREEN] {< wx1,wy1 > } - {< wx2,wy2 > }                                                   |
| WRITE        | < list of expressions >                                                                    |
| WRITE#       | < file number >,< list of expressions >                                                    |
| ?            | {< list of expressions > }                                                                 |

## FUNCTIONS

| Function                      | Example                   |
|-------------------------------|---------------------------|
| ABS(X)                        | Y = ABS(A + B)            |
| ASK(X)                        | PRINT ASK(X)              |
| ATN(X)                        | PRINT ATN(A)              |
| CDBIT(X)                      | A# = CDBIT(Y)             |
| CHR\$(I)                      | PRINT CHR\$(48)           |
| CINT(X)                       | B = CINT(B)               |
| COS(X)                        | A = COS(2.3)              |
| COSG(X)                       | C = COSG(X)               |
| CSRLIN                        | X = CSRLIN                |
| CVD(X < 8-byte string >)      | CH = CVD(X)               |
| CVI(X < 2-byte string >)      | Y# = CVI(X)               |
| CVS(X < 4-byte string >)      | A = CVS(X)                |
| DATES                         | PRINT DATES               |
| ENVIRON\$( < parm >   < n > ) | PRINT ENVIRON\$( "PATH" ) |
| EOH( < file number > )        | IF EOH(1) GOTO 300        |
| ERDEV                         | IF ERDEV = 57 THEN        |
| ERDEV\$                       | PRINT ERDEV\$             |
| ERL                           | PRINT ERL                 |
| ERR                           | IF ERR = 62 THEN          |
| EXP(X)                        | B = EXP(X)                |
| FIX(X)                        | I = FIX(A/B)              |
| FREQ()                        | PRINT FREQ()              |
| FREQ\$(X)                     | PRINT FREQ\$(X)           |
| HEX\$(X)                      | H\$ = HEX\$(100)          |

| Function                      | Example                                    |
|-------------------------------|--------------------------------------------|
| INKEY\$                       | A\$ = INKEY\$                              |
| INKEY                         | C = INKEY                                  |
| INPUT\$(X) ( # )              | Y\$ = INPUT\$(4) or<br>Y\$ = INPUT\$(5 #2) |
| INSTR(I) Y\$ Y\$)             | IF INSTR(Y\$ Y\$) = 0 THEN                 |
| INT(X)                        | C = INT(X + 3)                             |
| LOCATE(I) ( < file number > ) | IF LOCATE(1) < 7166 THEN                   |
| LEFT\$(X, I)                  | B\$ = LEFT\$(X, 6)                         |
| LEN(X)                        | PRINT LEN(B\$)                             |
| LOC( < file number > )        | PRINT LOC(1)                               |
| LOF( < file number > )        | IF LOF(2) > 5 THEN                         |
| LOG(X)                        | D = LOG(Y-2)                               |
| LPOS(X)                       | IF LPOS(1) > 60 THEN                       |
| MID\$(X, I, J)                | A\$ = MID\$(X, 5, 10)                      |
| MKDIR( < dir path exp > )     | D\$ = MKDIR( 67 / D# )                     |
| MKIS( < integer exp > )       | ISET A\$ = MKIS(B#)                        |
| MKSF( < 80 prec exp > )       | LSET D\$ = MKSF(A)                         |
| OCT\$(X)                      | PRINT OCT\$(100)                           |
| PEEK(X)                       | PRINT PEEK( &H2000 )                       |
| PEND( < n > )                 | IF PEND() = 0 THEN                         |
| PLAY( < n > )                 | I = PLAY(4)                                |
| PMAP( < x > , < n > )         | X = PMAP(1-1)                              |
| POINT( < x > , < y > )        | IF POINT(ROWCOL) < 0                       |
| POINT( < n > )                | POINT(2) < 319 THEN                        |
| POS()                         | IF POS(3) > 60 THEN                        |
| RIGHT\$(X, I)                 | C\$ = RIGHT\$(X, 8)                        |
| RND( < x > )                  | E = RND(1)                                 |

| Function                       | Example              |
|--------------------------------|----------------------|
| SCREEN (<row>,<br><col> [<Z>]) | X=SCREEN(10,20)      |
| SIN(X)                         | B=SIN(X+Y)           |
| SIN(X)                         | B=SIN(A)             |
| SPACES(X)                      | SS=SPACES(20)        |
| SPC(I)                         | PRINT SPC(15), A\$   |
| SQR(X)                         | C=SQR(D)             |
| STICK (<n>)                    | I=STICK(0)           |
| STR(X)                         | PRINT STR(35)        |
| STRIC (<n>)                    | IF STRIC(0)=0 THEN   |
| STRING(I)                      | Y\$=STRING(100,42)   |
| STRING(LX\$)                   | X\$=STRING(100,"A")  |
| TAB(I)                         | PRINT TAB(20)A\$     |
| TAN(X)                         | D=TAN(3.14)          |
| TIMES                          | PRINT TIMES          |
| TIMER                          | X=TIMER              |
| USR (<digit> [X])              | X=USR(2Y)            |
| VAL(X\$)                       | TOTAL=VAL(A\$)       |
| VARPTR (<var name>)            | I=VARPTR(X)          |
| VARPTR# (<file number>)        | J=VARPTR#2           |
| VARPTR\$ (<variable>)          | DRAW "X"+VARPTR(A\$) |

# C

## Assembly Language Subroutines

### Introduction

This appendix is provided for users who call assembly-language subroutines from their Vectra BASIC programs. If you do not use assembly-language subroutines, you may omit reading this appendix.

The USR function allows assembly-language subroutines to be called in the same way that Vectra BASIC intrinsic functions are called. However, we recommend that you use the CALL or CALLS statement for interfacing machine-language programs with Vectra BASIC. These statements produce more readable source code and can pass multiple arguments. In addition, the CALL statement is compatible with more languages than the USR function.

### Appendix C:

#### Assembly Language Subroutines

|      |                   |   |
|------|-------------------|---|
| C-1  | Introduction      | 4 |
| C-2  | Memory Allocation |   |
| C-3  | CALL Statement    |   |
| C-11 | USR Function      |   |

## Memory Allocation

You must set aside memory space for an assembly-language subroutine before you can load it. You accomplish this through the `/M:` switch in the Vectra BASIC command line. (The `/M:` switch sets the highest memory location that Vectra BASIC uses.)

In addition to the Vectra BASIC Interpreter code area, Vectra BASIC uses up to 64K of memory beginning at the Data Segment (DS).

When calling an assembly-language subroutine, if you need more stack space, you can save the Vectra BASIC stack and set up a new stack for the assembly-language subroutine. You must restore the stack, however, before the program returns from the subroutine.

You can load an assembly-language subroutine into memory through the operating system, the `POKE` statement, or by moving the code into a numeric or string array. If you have the software package for your microprocessor, routines may be assembled with the MACRO Assembler and linked, but not loaded, using the LINK Linking Loader. To load the program file, observe these guidelines:

- Make sure the subroutines do not contain any long references
- Skip over the first 512 bytes of the MS-LINK output file, then read in the rest of the file

## CALL Statement

The `CALL` statement is the recommended way of interfacing machine-language programs with Vectra BASIC. Do not use the `USR` function unless you are running previously written programs that already contain `USR` functions.

**Format:** `CALL variable: name (argument list) )`

**Remarks:** *variable: name* contains the segment offset that is the starting point in memory of the subroutine that you are calling.

*argument list* contains the variables or constants that are passed to the routine. You must separate the items in the list with commas.

Invoking the `CALL` statement causes the following events:

- For each parameter in the argument list, the 2-byte offset of the parameter's location within the Data Segment (DS) is pushed onto the stack.
- The Vectra BASIC return address code segment (CS) and offset (IP) are pushed onto the stack.
- Control is transferred to your routine through a long call to the segment address given in the last `DEF SEG` statement and the offset given in *variable: name*.

The following table illustrates the state of the stack at the time the CALL statement executes

|                |                                               |                                                |
|----------------|-----------------------------------------------|------------------------------------------------|
| High addresses | Parameter 0<br>Parameter 1<br><br>Parameter n | Each parameter is a 2-byte pointer into memory |
| Stack counter  | Return segment address                        |                                                |
| Low addresses  | Return offset                                 |                                                |
|                |                                               | Stack pointer (SP) register contents           |

Your routine now has control. You may refer to parameters by moving the stack pointer to the base pointer, then adding a positive offset to the base pointer.

The following figure shows the condition of the stack during execution of the called subroutine.

|                |                                               |                                                                 |
|----------------|-----------------------------------------------|-----------------------------------------------------------------|
| High addresses | Parameter 0<br>Parameter 1<br><br>Parameter n | Absent if any parameter is referenced within a nested procedure |
|                | Return segment address                        | Absent in local procedure                                       |
|                | Return offset                                 |                                                                 |
|                |                                               | Stack pointer (SP) register contents                            |

|               |                                                   |                                                     |
|---------------|---------------------------------------------------|-----------------------------------------------------|
| Stack counter | Local variables                                   | Only in reentrant procedure                         |
|               | This space may be used during procedure execution | Stack pointer may change during procedure execution |
| Low addresses |                                                   |                                                     |

The following rules apply when coding a subroutine:

1. The called routine may destroy the AX, BX, CX, DX, SI, and DI registers.
2. The called program must know the number and length of the parameters passed. References to parameters are positive offsets to BP (assuming the called routine moved the current stack pointer into BP).
3. The called routine must do a RET *n* statement, where *n* is twice the number of parameters in the argument list. This statement adjusts the stack to the start of the calling sequence.
4. Values are returned to Vectra BASIC by including a variable name in the argument list to receive the result.
5. If the argument is a string, the parameter's offset points to three bytes, which, as a unit, is called the **string descriptor**.

Byte 0 of the string descriptor contains the length of the string. This number may vary from 0 (if all 8 bits are zero) to 255 (if all 8 bits are ones).

Bytes 1 and 2, respectively, are the lower and upper 8 bits of the starting string address in string space.

### Caution

If the argument is a string literal in the program, the string descriptor points to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add `*"` to the string literal in the program. For example, the following statement forces the string literal to be copied into string space:

```
20 A$ = "BASIC" * " "
```

You may now modify this string without affecting the program.

6. Strings may be altered by user routines, but their length **MUST REMAIN THE SAME**. Vectra BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

### Example:

The following program loads an assembly-language subroutine into memory, then calls it. The DATA statements contain the assembled code with byte pairs inverted. (This is an easy way of loading the code into memory.) The call to `VARPTR` locates the starting location of the first byte of the code, then the subroutine is called using that offset. It is important to include the `VARPTR` call just prior to the subroutine call since you need the current location and the array containing the code may move around in memory as Vectra DAS/C defines more variables.

```
10 REM GET CURRENT DS-REGISTER VALUE
20 DATA &H8B55,&H8BEC,&H065E,&HD8BC
30 DATA &H0789,&HCASD,&H0002
40 DIM GETDS%(6): FOR IND% = 0 TO 6
50 READ GETDS%(IND%): NEXT IND%
60 ADDR% = VARPTR(GETDS%(0)): CALL ADDR%(&ADDR%)
70 PRINT HEX$(ADDR%)
```

A copy of the assembly-language subroutine follows:

|               |               |                                            |
|---------------|---------------|--------------------------------------------|
| 0000          | code          | segment byte public 'code'                 |
|               |               | public gets                                |
|               |               | assume cs:code                             |
|               |               | ; subroutine: return DS to calling program |
| 0000          | gets          | proc far ; Start of procedure              |
| 0000 55       | push bp       | ; Save current bp register                 |
| 0001 8B EC    | mov bp,sp     | ; Use bp as stack pointer                  |
| 0003 8B 5E 06 | mov bx,[bp+6] | ; Load addr of variable into bx            |
| 0006 8C D8    | mov ax,ds     | ; Load value of DS into ax                 |
| 0008 89 07    | mov [bx],ax   | ; Store value of DS (in ax) into variable  |
| 000A 5D       | pop bp        | ; Recall original value of bp              |
| 000B CA 0002  | ret 2         | ; Return to main prog. with cleanup        |
| 000E          | ends          | ; End of procedure                         |
| 000E          | code          |                                            |
|               |               | end                                        |

The following sequence in assembly-language code demonstrates access of the parameters passed. The return result is stored in variable 'C'.

```

PUSH BP
MOV BP,SP
MOV BX,[BP+8]
MOV CL,[BX]
MOV DX,[BX+1]

;Save BP register
;Get current stack position in BP
;Get address of BS slope
;Get length of BS in CL
;Get address of BS text in DX

MOV SI,[BP+10]
MOV DI,[BP+6]
; MOVS WORD
; POP BP
RET 6
;Get address of 'A' in SI
;Get pointer to 'C' in DI
;Store variable 'A' in 'C'.
;Restore BP register
;Restore stack, return

```

### Note

The called program must know the variable type for the numeric parameters passed. In the previous example, the instruction MOVSW copies only 2 bytes. This suffices when variables A and C are integers. However, you have to copy 4 bytes if the variables are single-precision values and 8 bytes if they are double-precision values.

## USR Function

Although the CALL statement is the recommended way of calling assembly-language subroutines, the USR function is still available for compatibility with previously written programs.

**Format:** `USR (digit) (argument)`

**Remarks:** *digit* is an integer that ranges from 0 to 9. It specifies which USR routine is being called and corresponds with the digit supplied in the DEF USR statement for that routine. If you omit *digit*, Vectra BASIC assumes the call is to USR0.

*argument* is any numeric or string expression.

In Vectra BASIC, you must execute a DEF USR statement before calling a USR function to ensure that the code segment points to the subroutine being called. The address given in the DEF SEG statement determines the starting address of the subroutine.

For each USR function, you must execute a DEF USR statement to define the USR function offset. This offset and the currently active DEF SEG statement determines the starting segment of the subroutine.

When the USR function call is made, register AL contains a value that specifies which type of argument was given. The value in AL may be one of the following:

| Value in AL | Type of Argument                       |
|-------------|----------------------------------------|
| 2           | Two-byte integer (two's complement)    |
| 3           | String                                 |
| 4           | Single-precision floating point number |
| 8           | Double-precision floating point number |

If the argument is a number, the BX register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

The Floating Point Accumulator is the exponent minus 128. (The radix point is to the left of the most significant bit of the mantissa.)

FAC-1 contains the highest 7 bits of the mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive; 1 = negative).

If the argument is an integer:

FAC-2 contains the upper 8 bits of the argument.

FAC-3 contains the lower 8 bits of the argument.

If the argument is a single-precision floating point number:

FAC-2 contains the middle 8 bits of the argument.

FAC-3 contains the lowest 8 bits of the argument.

If the argument is a double-precision floating point number:

FAC-7 through FAC-4 contain four more bytes of the mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the DX register pair points to three bytes. These three bytes are called the **string descriptor**.

Byte 0 contains the length of the string. This value varies from 0 (if all 8 bits are zeros) to 255 (if all 8 bits are ones).

Bytes 1 and 2, respectively, are the lower and upper eight bits of the starting string address in the Data Segment.

### Caution

If the argument is a string literal in the program, the string descriptor points to program text. Be careful not to alter or destroy your program this way.

Usually, the value returned by a **USR** function is the same type (integer, single-precision, double-precision, or string) as the argument that was passed to it.

### Example:

```
100 DEF USR0=4H800 'Assumes /M:32767
120 X = 5
130 Y = USR0
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed.

## Appendix D:

### Installing Vectra BASIC

- D-1** Making a Working Copy of Vectra BASIC
- D-2** For Dual Disc Drive Users
- D-3** Copying Vectra BASIC
- D-4** Making Vectra Basic a P.A.M. Option
- D-5** For Hard Disc Drive Users
- D-6** Starting Vectra BASIC

## D

### Making A Working Copy Of Vectra BASIC

You should always make a working copy of your application software as a safeguard against possible damage or loss. Since the Vectra supports a variety of peripheral mass storage devices, the actual procedure depends upon which disc drive you are using. The following sections describe making a working copy of Vectra BASIC using either a dual disc drive or a hard disc drive. As the system directs you on each step you must take, you may follow the instructions on the screen if you have a different type of disc drive.

For users with pre-existing BASIC applications and .BAT files, your Vectra BASIC disc contains **BASICA .COM** and **BASIC .COM** for your convenience. You should copy them when you copy **GW BASIC**.

#### Caution

Before going through the install procedure, you should write-protect your master disc to prevent any accidental "over-writing". For information on write-protecting your disc, refer to the owner's manual that accompanied your disc drive.

## Appendix D

## For Dual Disc Drive Users

The following discussion lists the steps that you should follow to make a working copy of your Vectra BASIC master disc. For this procedure, you need the following discs:

- Your working copy of the HP Vectra DOS Disc
- Your working copy of the SUPPLEMENTAL DISC
- Your master copy of Vectra BASIC
- An unformatted disc

Your computer assumes drive A (the top drive) is the currently active drive, unless you have taken steps to instruct it differently. This procedure, therefore, requires your inserting the "controlling" discs into drive A.

Inserting a disc into a drive is an easy task:

- Hold the disc by its label end.
- Inspect both sides of the disc. You can recognize the top since it has printing on the shutter and also contains the larger portion of the label. The most obvious feature on the bottom is the circular head.
- Ensure that the top of the disc (the labeled side) is facing up when you insert the disc into a drive.

## Copying Vectra BASIC

1. Insert your working copy of the Vectra DOS Disc (the one containing "P.A.M." into drive A.
2. Put a blank disc in drive B.
3. Do a System Reset (by simultaneously pressing **CTRL** **[Alt]** **[Sys req]**) to put the system in its initial, power-on state.
4. To format the disc and copy P.A.M. onto it, type **FORMAT B: /P**. If you want to add your own Volume label, type **FORMAT B: /P/V**. When the formatting process is complete, you will be prompted for a Volume label. Type **VBASIC** (or a name of your choice) and press **[Enter]**. (If you don't wish to include P.A.M. type **FORMAT B: /S**, or **FORMAT B: /V/S**.)
5. When the formatting process is complete, **FORMAT** will ask the question "Format another?". Type **N** **[Enter]**, then hit any key to return to the P.A.M. menu.
6. Remove your DOS Disc from drive A and insert your Vectra BASIC master disc in that drive.
7. Type **COPY A: GWBASIC.EXE B:**.

You have now successfully installed Vectra BASIC on your disc. Put your Vectra BASIC master disc in a safe place and use your working copy.

## Making Vectra BASIC a P.A.M. Option

If you have included P.A.M. on your working copy of Vectra BASIC and you wish to add Vectra BASIC to your P.A.M. menu, follow these steps:

1. Insert your working copy of P.A.M. and Vectra BASIC in drive A, and press function key **F5** for **Manage Applications**.
2. Press **F1** to select **Add**.
3. Press **F5** to select **Add Unlisted**.

## For Hard Disc Drive Users

This section details the steps that you must take to place a working copy of Vectra BASIC on a hard disc.

For this procedure, you need the following discs:

- Your back-up copy of the HP Vectra DOS Disc
- Your back-up copy of the Supplemental DISC
- Your master copy of Vectra BASIC
- Your hard disc drive

1. If you have not already done so, format your hard disc. (The owner's manual for your hard disc supplies the necessary details.)
2. Put your working copy of the DOS Disc (the disc containing P.A.M.) into the flexible disc drive A and bring up the main P.A.M. menu.
3. If you want to place Vectra BASIC in a subdirectory on your hard disc, create that subdirectory, if it doesn't exist. (Just type `MKDIR C:\pathname`.)
4. If you are placing Vectra BASIC in a subdirectory, change to that directory by typing `CD C:\pathname`.
5. Remove your copy of the DOS Disc from drive A, and insert your Vectra BASIC master disc.
6. Type `COPY A:GW BASIC.EXE C:`.

If you want to install Vectra BASIC on your P.A.M. menu, press function key `F5` from the P.A.M. Main Menu. Then follow the steps 2-6 in the section above, Making Vectra BASIC a P.A.M. option. Be sure to name the hard disc (and the appropriate pathname) in the Path field as part of Step 4.

4. The path name does not need to be changed, so press `[Enter]`.  
In the Application Title field, type `Vectra BASIC`, then press `[Enter]`.  
In the Run Command field, type `GW BASIC`, then press `[Enter]`.

### Note

If you're going to be using Vectra BASIC with any of the command line switches or Input/Output Redirect, these should be included in the Run Command. For example, to include a /M switch, the Run Command might read `GW BASIC /M 32000`. (See Chapter 3 for more information.)

5. Press `F1` to **Save** this information.

If you want Vectra BASIC to start automatically when you start your system, perform the steps below. Otherwise, skip to step 6.

To AUTOSTART Vectra BASIC:

- a) Press `F8 (EXIT)` twice to return to the Manage Applications menu.
- b) Press `F6` to select `Auto Start`.
- c) Move the arrow to Vectra BASIC and press `F1` to **Save**.
- d) Test this procedure by simultaneously pressing `[CTRL] [Alt] [DEL]`. Vectra BASIC should now start automatically.

6. If you don't want to make Vectra BASIC an Auto Start application, just press `F8 (EXIT)` 3 times to return to the Main Menu. Now select Vectra BASIC by moving the arrow, and press `F1` to **Start Application**. Vectra BASIC should start.

## Starting Vectra BASIC

With a flexible disc drive, after you have both the operating system and Vectra BASIC on a single disc, running Vectra BASIC becomes an easy task. Simply start your system with this disc in the drive, and select Vectra BASIC from the P.A.M. Main Menu and press function key f1 for **Start Application**.

If you have started up your system with a different P.A.M. disc, insert your Vectra BASIC disc in drive A, then press f6 for **Show .EXE .COM .BAT**. Move the arrow to **GBASIC.EXE** and press **Enter**.

Once you have Vectra BASIC installed on your hard disc drive, simply select Vectra BASIC from the P.A.M. main menu and press **Start Application**.

Chapter 3 tells how you may increase your programming flexibility when entering Vectra BASIC.

# E

## Differences in Versions of BASIC

This appendix documents the differences between Vectra BASIC and other implementations of Microsoft's GW BASIC interpreter.

### Math rounding and random numbers

Certain small differences in mathematical precision exist in different versions of BASIC. The differences are extremely small, in the range of 1E-7.

### Screen editor and keyboard functions

In Vectra BASIC, the Alt-M key combination produces the BASIC word **MERGE**. Other versions which support cassette recorder operation produce the word **MOTOR**.

### Command line options

Vectra BASIC uses Microsoft's dynamic memory allocation for File Control Block space. The /1, /S and /F command line options make it possible to emulate the static memory allocation of other versions of BASIC. See Chapter 3 for more information.

### Input/Output redirection

When output is redirected in Vectra BASIC, using **CTRL-Break** to stop program execution returns control to BASIC, rather than to DOS. All subsequent screen output is written to the output file as well as to the screen. Other versions of BASIC write an error message to the output file, then exit to DOS.

### SHELL

The **SHELL** command in Vectra BASIC allows you to run BASIC as a child of BASIC. In some versions of BASIC, you cannot **SHELL** to BASIC. Also, in some versions of BASIC, any text which remains on the screen following a **SHELL** process is "visible" to the screen editor. In Vectra BASIC, this text is not recognized by the screen editor.

**File Control Blocks**

The Vectra BASIC File Control Block is slightly different than the File Control Block in other versions of BASIC.

**ERDEV**

Some versions of BASIC include device attribute bits as well as the Interrupt X24 code in **ERDEV**. Vectra BASIC reports only the Interrupt X24 code.

**GOSUB nesting**

Vectra BASIC allows **GOSUB** nesting up to 32 levels. Some versions of BASIC allow 1 more level of nested **GOSUBS**.

**OPEN "COM Parity checking**

When using the **OPEN "COM** statement, Vectra BASIC always performs parity checking. In some versions of BASIC, parity checking is only performed if the **PE** option is specified. In Vectra BASIC, the **PE** option has no effect, but may be included on the command line to maintain compatibility.

**Trailing blanks on input**

Vectra BASIC strips trailing blanks from all **INPUT** statements, including **INPUT** statements which read from files. This is compatible with the ANSI standard. Some versions of BASIC do not delete these trailing blanks.

**Number of open files**

By default, Vectra BASIC allows five files to be open simultaneously. Other versions of BASIC may allow a different number of open files.

**WIDTH settings**

Vectra BASIC allows the **SCRN**: device to be set to any **WIDTH** from 1 to 255. Some versions restrict the **SCRN**: device to 40 and 80 columns.

**Renaming directories**

Vectra BASIC does not allow renaming of directories with the **NAME** statement. Renaming directories is allowed by some BASIC versions.

**STRIG ON and OFF**

Vectra BASIC treats **STRIG ON** and **OFF** slightly differently than other versions. With **STRIG OFF** in effect, Vectra BASIC does not retain previous trigger values. When **STRIG ON** is in effect, previous and current values are retained.

**E-2 Differences in Versions of BASIC**

**PEN reading**

Currently, **PEN (0)** always returns 0.

**VIEW PRINT**

The **VIEW PRINT** command, which opens a text window, is a new Microsoft GW BASIC feature. It is not supported in earlier BASIC versions.

**Last referenced graphics point**

When a Control-L or Control-Home keystroke combination is used to clear the screen while a **VIEW** is in effect, Vectra BASIC resets the last referenced graphics point to the center of the viewport. Other versions of BASIC reset the last referenced graphics point to the center of the screen, or some other position. Graphics commands using relative positions (**STEP**) will perform differently.

**Error message differences**

Certain error conditions in Vectra BASIC return different error messages than other BASIC versions. The only circumstance where this could have serious effects is in situations which would invoke error trapping routines that depend on the error number. Three of these cases are:

**OPEN "COMn...** with no serial port installed reports  
**Bad file name** in Vectra BASIC; and  
**Device unavailable** in some versions.

Exceeding the allowed number of **OPEN** files results in a **Too many files** message in Vectra BASIC; other BASICs report **Bad file number**.

**SCREEN** with no arguments produces an **Illegal function call** error message in Vectra BASIC, and a **Missing operand** error in other versions.

## Index

### A

ABS 6-3  
Absolute Value 6-3  
Adding Text 1-14  
Algebraic Expressions 2-13  
Alphabetizing Strings 2-20  
Alt Key 1-8  
Altering Data And Variables 5-8  
AND 2-15, 6-242  
Animation 6-209, 6-243  
Arctangent 6-4  
Arithmetic Functions 5-26  
Arithmetic Functions, Derived 5-27  
Arithmetic Operators 2-12  
Arithmetic Overflow 2-14  
Array Variables 2-6  
Arrays, Deleting 6-82  
Arrays, Dimensioning 6-66  
Arrays, Initial Subscript 6-201  
ASC 6-3  
ASCII Character Codes B-3  
Aspect Ratio 6-28  
Assembly Language Subroutines C-1  
Assigning Values To Variables 6-140  
Asterisk After Line Number 6-5  
ATN 6-4  
AUTO 6-5

## B

**D-3** Back-up Copy For Vectra BASIC  
**5-204** Background Tile Pattern  
**6-197** Baud Rate  
**6-7** BEEP  
**6-7** Bell Control  
**2-17** Bits, Masking  
**2-17** Bits, Merging  
**6-203** Blacking-out An Area  
**6-336** Blinking Characters  
**6-6** BLOAD  
**6-71, 6-141** Box, Drawing  
**xiv** Braces  
**xiv** Brackets  
**5-13** Branching Statements  
**5-13** Branching To Another Program  
**5-13** Branching, Conditional  
**5-13** Branching, Unconditional  
**6-11** BS/VE  
**6-319** Bubble Sort

## C

**6-13, C-3** CALL  
**6-16** CALLS  
**xiv** Capital Letters  
**6-17** CDBL  
**6-18** CHAIN  
**6-35, C-1** Changing Character Display  
**6-23** Changing Directories  
**6-2** Chapter Format  
**2-20** Character Comparisons  
**6-35, C-1** Character Enhancements  
**1-6, B-1** Character Sets  
**6-23** CHDIR  
**6-25, B-2** CHR\$  
**6-26** CINT

**6-27** CIRCLE  
**6-31** CLEAR  
**6-33** Clearing The Screen  
**6-279** Clipping  
**6-33** CLOSE  
**6-33** CLS  
**1-5** Colon As Statement Separator  
**6-36** COLOR (graphics mode)  
**6-35** COLOR (text mode)  
**6-160** Column Position  
**6-41** COM(n)  
**1-2** Command Level  
**5-5** Commands Used As Program Statements  
**6-42** COMMON  
**5-9** Computer Control Statements  
**2-20** Concatenation  
**5-15** Conditional Branching Statements  
**2-2** Constants  
**6-45** CONT  
**1-15, 1-17** Control Characters  
**6-5** Control-Break, Cancelling AUTO  
**6-116** Control-Break, Cancelling INKEY\$  
**6-144** Control-Break, Cancelling LINE INPUT  
**6-149** Control-Break, Cancelling LIST  
**6-317** Control-Break, Cancelling WAIT  
**1-17, 6-295** Control-Break, Returning To Command Level  
**3-8** Control-Break, Returning To MS-DOS  
**6-47** COS  
**6-27** Coscant  
**5-27** Cotangent  
**6-166** Creating A Directory  
**6-48** CSNG  
**6-49** CSRLIN  
**4-1** Current Directory  
**6-67** Current Graphics Position  
**6-50** CVD  
**6-50** CVI  
**6-50** CVS

## D

**6-51** DATA  
**6-197** Data Bits  
**2-1** Data Operators  
**6-51, 6-253** DATA Statements, Rereading  
**2-1** Data Variables  
**6-53** DATES Function  
**6-54** DATES Statement  
**5-22** Debugging Statements  
**2-4, 2-5** Declaration Characters  
**6-56** DEF FN  
**6-58** DEF SEG  
**6-60** DEF USR  
**6-62** DEFDBL  
**6-62** DEFINT  
**6-62** DEFNG  
**6-62** DEFSTR  
**5-8** Defining Data Or Variables  
**6-88** Defining Error Codes  
**6-64** DELETE  
**1-13** Deleting Text  
**4-2** Depth Of A Path  
**5-24** Device Sampling Functions  
**5-27** Derived Functions  
**E-1** Differences Between Vectra BASIC and other versions  
**6-66** DIM  
**1-2** Direct Mode  
**6-23** Directories, Changing  
**6-168** Directories, Creating  
**6-257** Directories, Removing  
**4-1, 5-6** Directory Operations  
**4-1** Directory Paths  
**4-3** Disc File Names  
**4-2** Disc File Naming Conventions  
**2-14** Division By Zero  
**1-18** Documenting Your Program  
**2-3** Double Precision  
**6-68** DRAW  
**6-71, 6-141** Drawing A Box  
**6-141** Drawing A Dotted Line

## E

**6-89** <sup>e</sup>  
**6-73** EDIT  
**1-15** Edit Keys  
**xlv** Ellipsis  
**6-110** ELSE  
**6-74** END  
**1-9** Entering A Program  
**6-75** ENVIRON  
**6-77** ENVIRON\$  
**6-79** EOF  
**6-81** EOF (For COM Files)  
**6-111** Equality Testing  
**2-17** EQV (Equivalent)  
**6-82** ERASE  
**1-13** Erasing Text  
**6-83** ERDEV  
**6-85** ERL  
**6-85** ERR  
**6-87** ERROR  
**A-1** Error Codes  
**6-88** Error Codes, Defining  
**1-18, A-1** Error Messages  
**2-11, 2-13** Evaluation Order  
**5-18** Event Trapping Statements  
**6-294** Exchanging Values  
**2-17** Exclusive OR  
**1-5** Executable Statement  
**6-89** EXP  
**2-2** Exponent, Floating Point  
**6-233** Exponential Format  
**2-12, 6-89** Exponentiation  
**2-12** Expressions  
**6-115** Extended Codes

## F

**6-105** "Falling Through"  
**6-90** FIELD  
**6-308** File Control Block  
**4-3** File Names  
**4-1, 5-6** File Operations  
**6-93** FILES  
**4-2** Filespec  
**6-95** FIX  
**2-2** Fixed Point Constants  
**2-2** Floating Point Constants  
**6-96** FOR  
**5-13, 6-96** FOR/NEXT Loops  
**6-2** Format For Functions  
**6-2** Format For Instructions  
**1-4** Formatting A Program Line  
**6-231** Formatting Numbers  
**6-230** Formatting Strings  
**6-90** Formatting The Random-tile Buffer  
**6-100** FRE  
**2-19** Functional Operators  
**5-23** Functions

## G

**5-24** General Purpose Functions  
**6-5** Generating Line Numbers Automatically  
**6-101** GET  
**6-104** GET (For COM Files)  
**6-102** GET (For Graphics)  
**6-105** GOSUB  
**6-106** GOTO  
**5-10** Graphics Statements  
**6-235** Graphics Screen "Jumping"  
**3-1** GWBASIC Command Line  
**3-1** GWBASIC

## H

**2-3** Hex Constants  
**6-109** HEX\$  
**B-1** HP Character Set  
**5-28** Hyperbolic Trigonometric Functions

## I

**6-110** IF  
**2-17** IMP (Implied)  
**2-16** Inclusive OR  
**1-4** Indirect Mode  
**6-317** Infinite Loop With WAIT  
**6-114** INKEY\$  
**6-117** INP  
**6-118** INPUT  
**3-16** Input Commands  
**1-11** Input Editing  
**3-7** Input, Redirecting  
**5-25** Input/Output Functions  
**6-122** INPUT#  
**6-124** INPUT\$  
**D-1** Installing Vectra BASIC  
**6-126** INSTR  
**6-127** INT  
**2-2** Integer Constants  
**2-14** Integer Division  
**5-27** Inverse Trigonometric Functions  
**6-35** Inverse-video Characters  
**6-128** IOCTL  
**6-129** IOCTL\$  
**xiv** Italicized Words

## J

**6-284** joystick  
**6-288** joystick triggers  
**6-162** Justifying Text

## K

**6-130** KEY  
**6-134** KEY(n)  
**6-136** KILL

## L

**6-132** Latched Keys  
**4-2** Leaf  
**6-136** LEFTS  
**6-162** Left-justifying A String  
**6-139** LEN  
**6-140** LET  
**6-184, 6-211, 6-212** Light pen  
**6-141** LINE  
**1-4** Line Format  
**6-145** LINE INPUT  
**6-146** LINE INPUT#  
**6-146** LIST  
**6-146** LJUST  
**6-151** LOAD  
**6-153** LOC  
**6-154** LOC (For COM Files)  
**6-155** LOCATE  
**6-156** LOF  
**6-159** LOF (For COM Files)  
**6-160** LOG  
**1-5** Logical Line  
**2-15** Logical Operators  
**5-15** Looping Statements  
xiv Lower Case Letters  
**6-160** LPOS  
**6-161** LPRINT  
**6-161** LPRINT USING  
**1-19, 6-148, 6-194,** LPT:  
**6-196** LPT:  
**6-162** LSET

## M

**2-14** Machine Infinity  
**D-3** Making A Backup Copy For Vectra BASIC  
**2-2** Mantissa, Floating Point  
**2-16** Masking Bits  
**C-2** Memory Allocation  
**6-6** Memory Image File  
**6-164** MERGE  
**2-16** Merging Bits  
**6-166** MID\$ Function  
**6-167** MID\$ Statement  
**6-166** MKDIR  
**6-170** MKID\$  
**6-170** MKIS  
**6-170** MKSS  
**2-14** MOD  
**6-197** Modem, programming  
**1-2** Modes Of Operation  
**1-11** Modifying Text  
**2-14** Modulus Arithmetic  
**1-12** Moving The Cursor  
**5-12** Music Statements

## N

**6-171** NAME  
**6-69, 6-160** Natural Logarithms  
**6-97** Nesting FOR Loops  
**6-112** Nesting IF Statements  
**6-105** Nesting Subroutines  
**6-316** Nesting WHILE Loops  
**6-173** NEW  
**6-96** NEXT  
**1-5** Non-executable Statement  
**2-16** NOT  
xiv Notation Conventions  
**6-231** Numeric Fields  
**2-5** Numeric Variables

## O

2-3 Octal Constants  
 6-174 OCT\$  
 6-175 ON COM  
 6-177 ON ERROR GOTO  
 6-179 ON...GOSUB  
 6-180 ON...GOTO  
 6-181 ON KEY  
 6-184 ON PEN  
 6-186 ON PLAY  
 6-188 ON STRIG  
 6-190 ON TIMER  
 6-192 OPEN  
 6-197 OPEN "COM"  
 2-12 Operators  
 6-201 OPTION BASE  
 2-16, 6-242 OR  
 2-11, 2-13 Order Of Precedence  
 6-202 OUT  
 5-16 Output Commands  
 5-25 Output Functions  
 3-7 Output, Redirecting  
 2-14 Overflow In Arithmetic Operations

6-220 POINT  
 6-222 POKE  
 6-223 POS  
 Preface  
 6-224, 6-242 PRESET  
 6-226 PRINT  
 1-19 Print Operations  
 6-229 PRINT USING  
 6-226 Print Zones  
 6-235 PRINT#  
 6-235 PRINT# USING  
 6-231 Printing Numbers  
 6-230 Printing Strings  
 5-13 Program Control Statements  
 1-4 Program Lines  
 1-1 Programming Guidelines  
 5-1 Programming Tasks  
 4-9 Protected Files  
 6-238, 6-242 PSET  
 Punctuation  
 6-240 PUT  
 6-104 PUT (For COM Files)  
 6-241 PUT (For Graphics)

## P

6-203 PAINT  
 6-325 Pan  
 3-4 Paragraphs  
 2-13 Parentheses And Order Of Evaluation  
 6-197 Parity  
 4-1 Path  
 4-2 Path Names  
 6-210 PEEK  
 6-211 PEN  
 6-212 PEN(n)  
 6-187 Physical Coordinates  
 6-214 PLAY  
 6-218 PLAY(n)  
 6-219 PMAP

## Q

1-13, 6-227 Question Mark  
 6-118, 6-145 Question Mark Prompt  
 6-118, 6-122 Question Mark Prompt, Suppressing  
 1-3 Quick Computation

## R

6-259 "Random" Numbers  
 4-4 Random-Access Files  
 6-244 RANDOMIZE  
 6-246 READ  
 D-3 Rebooting The System  
 4-4 Records  
 3-7 Redirecting Standard I/O

**B-1** Reference Tables  
**2-15** Relational Operators  
**6-248** REM  
**6-257** Removing Directories  
**6-250** RENUM  
**6-51, 6-253** Rereading DATA Statements  
**2-4, B-17** Reseeding Random-number Generator  
**6-252** Reserved Words  
**6-253** RESET  
**6-20** RESTORE  
**6-254** RESTORE With CHAIN  
**6-255** RESUME  
**6-256** RETURN  
**6-162** RIGHTS  
**6-257** Right-justifying A String  
**6-259** RMDIR  
**4-1** RND  
**4-1** Root  
**6-49** Row Position  
**6-162** RSET  
**5-17** RS232 Asynchronous Communication Statements  
**6-260** RUN

## S

**6-262** SAVE  
**6-314, 6-315** Scaling  
**B-14** Scan Codes  
**6-264** SCREEN Function  
**1-8** Screen Line Editor  
**6-267** SCREEN Statement  
**5-27** Secant  
**4-3** Sequential Files  
**6-272** SGN  
**6-273** SHELL  
**6-276** SIN  
**2-3** Single Precision  
**6-277** SOUND

**1-12** Space Bar  
**6-281** SPACES\$  
**6-282** SPC  
**5-30** Special Functions  
**xiv** Square Brackets  
**6-283** SQR  
**6-27, 6-141, 6-224, 6-238** STEP Option  
**6-96** STEP With FOR Statement  
**6-284** STICK  
**6-285** STOP  
**6-197** Stop Bits  
**6-287** STR\$  
**6-288** STRIG  
**6-289** STRIG(n) Function  
**6-291** STRIG(n) Statement  
**6-230** String Fields  
**5-29** String Functions  
**2-20** String Operations  
**2-20** String Operators  
**2-5** String Variables  
**6-293** STRING\$  
**5-15** Subroutine Statements  
**6-294** SWAP  
**B-18** Syntax Diagrams  
**6-295** SYSTEM  
**5-3** System Commands  
**D-3** System Reset

## T

**6-296** TAB  
**6-297** TAN  
**5-16** Terminal I/O Statements  
**6-111** Testing Equality  
**6-110** THEN  
**6-204** Tile Mask  
**6-298** TIMES Function  
**6-299** TIMES Statement  
**6-300** TIMER Function  
**6-301** TIMER Statement

**6-303** Trace Flag  
**6-303** TROFF  
**6-303** TRON  
**2-16** Truth Tables  
**2-18** Two's Complement  
**2-8** Type Conversion  
**2-4, 6-62** Type Declaration Characters

**U**  
**6-130** User-defined Function Keys  
**6-60** User-Defined Functions  
**6-35** Underlining Characters  
**5-5** Using Commands As Program Statements  
**6-305, C-11** USB Function

**V**  
**6-306** VAL Function  
**6-200** Variable Length String Field  
**2-4** Variables  
**6-307** VARPTR  
**6-310** VARPTR\$  
**5-23** Vectra BASIC Functions  
**zlv** Vertical Bar (|)  
**6-312** VIEW  
**6-316** VIEW PRINT

**W**  
**6-317** WAIT  
**6-318** WHILE...WEND  
**6-203** Whiting-out An Area  
**6-320** WIDTH  
**6-93, 6-135** Wild Cards  
**6-322** WINDOW  
**6-325** Window Clipping

**1-13** Word  
**6-219** World Coordinates  
**6-326** WRITE  
**6-327** WRITE#  
**1-20** Writing A Simple Program

**X**  
**2-17, 6-242** XOR

**Z**  
**6-325** Zoom